Broadview®
www.broadview.com.cn

疯狂 Spring Cloud

微服务架构实战

杨恩雄 编著

疯狂源自梦想
技术成就辉煌

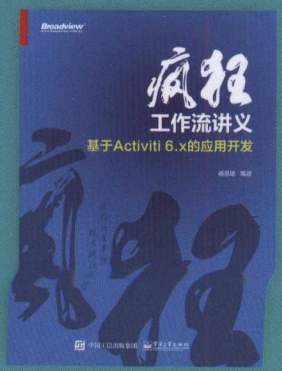
扫码关注
“疯狂图书”微信号可获得：
· 本书配套源代码
· 价值158元的相关视频
· 作者不定期答疑服务

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介

杨恩雄，从事十多年Java EE企业应用开发，曾任中企动力系统设计师、中企开源项目经理、数码辰星科技公司项目经理，参与过多个企业级项目的设计与架构工作，曾负责辰星“电影票网络销售系统”的整体架构。他精通Activiti、Drools、ESB等开源技术，在SOA、SaaS、大数据应用、互联网系统架构方面有丰富的经验，还出版了《疯狂Java实战演义》《疯狂Workflow讲义》等书籍。

其他作品



疯狂 Spring Cloud

微服务架构实战

杨恩雄 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书以 Spring Cloud 为基础，深入讲解微服务开发的相关框架，包括服务管理框架 Eureka、负载均衡框架 Ribbon、服务客户端 Feign、容错框架 Hystrix、消息框架 Stream 等。除了介绍这些微服务相关的框架外，在本书的第 11 章，还介绍了如何使用 Spring Data 框架操作各个主流数据库（MySQL、MongoDB、Redis）。在第 12 章，以一个案例为基础结束本书内容，在该章中讲解了模板引擎 Thymeleaf，整本书将会为大家提供一整套微服务应用开发的解决方案。

本书适合有一定 Java 开发基础的技术人员，尤其是正在使用或准备使用微服务构建高并发、大数据应用的技术人员及团队。

图书在版编目（CIP）数据

疯狂 Spring Cloud 微服务架构实战 / 杨恩雄编著. —北京：电子工业出版社，2018.1
ISBN 978-7-121-33109-1

Ⅰ. ①疯… Ⅱ. ①杨… Ⅲ. ①互联网络－网络服务器 Ⅳ. ①TP368.5

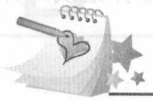
中国版本图书馆 CIP 数据核字（2017）第 288307 号

策划编辑：张月萍
责任编辑：刘　舫
印　　刷：北京京科印刷有限公司
装　　订：北京京科印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：18.5　　字数：370 千字
版　　次：2018 年 1 月第 1 版
印　　次：2018 年 1 月第 1 次印刷
印　　数：3000 册　　定价：58.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。



序

Java语言经过多年的发展，从开始的EJB + Servlet的开发模式，到后来的轻量级Java EE开发模式，每一种架构或开发模式的出现，都牵动着万千程序员的心。程序员是一个求知若渴的群体，也是一个容易接受新知识的群体，为了学习新技术，多少人食不知味、夜不能寐。笔者有幸成为一名平凡的程序员，从业十余年，面对各种新技术，时常怀着一颗谦卑的心不停前进，只有学习，才能带来快乐，才不会被淘汰。

笔者大约从2010年开始接手高并发、大数据的系统，经历过项目重构、人肉运维等痛苦过程，深知项目架构之不易，一直以来，期望能构建出完美的系统。此次恰逢Spring Cloud的兴起，笔者亦难以北窗高卧，数月以来寝食不安，编纂拙著。笔者学识浅薄，加之仓促付梓，书中错漏定然难免，望读者见谅。

IT行业人才辈出，笔者为平庸之辈，今日有幸与大家分享知识，实乃上天的眷顾。本书能得以出版，得益于多方襄助，对他们的感激之情，难以言表。感谢传道并解惑的恩师，感谢聪颖而好学的读者，感谢善良且亲爱的家人，笔者会谢意永存、铭感不忘。

本书特点

笔者长期工作于企业的IT部门，有着丰富的企业应用开发经验，因此本书具有以下特点。

1. 内容深入

从笔者接触编程开始，就养成了查看源代码的习惯，书中不仅仅讲解Spring Cloud的功能，更深入Spring Cloud的原理。

2. 案例详细

本书的每个知识点，几乎都会对应一个案例。在本书最后，还附有一个完整的案例，读者在该案例基础上，可建立自己的项目。

衷心感谢

首先非常感谢李刚老师，一直以来，他既是我的老师，也是我的技术后盾，非常幸运人生能有这样一位良师益友。

其次感谢出版社编辑，不辞辛苦地为我的书纠正各种错误，并为本书提出了许多宝贵的意见。

最后感谢我的家人，你们是我前进的动力。

本书写给谁看

如果你有一定的 Java 语言基础，进行过 Web 项目的开发，那么本书可以为你带来一个全新的开发模式。如果你是一名系统设计师，本书可以让你学习全新的系统架构。如果你是一名维护系统的程序员，即使本书的架构不适合你的系统，但本书的技术框架，仍然可以为你的系统改造和完善提供参考。

个人简介

本人从事十多年的 Java EE 企业应用开发，曾任中企动力系统设计师、中企开源项目经理、数码辰星科技公司项目经理，参与过多个企业级项目的设计与架构工作。曾负责辰星“电影票网络销售系统”的整体架构，精通 Activiti、Drools、ESB 等开源技术，在 SOA、SaaS、大数据应用、互联网系统架构方面有着丰富的经验，曾出版《疯狂 Java 实战演义》《疯狂 Workflow 讲义》等书籍。

杨恩雄

2017 年 10 月

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书如提供示例代码及资源文件，均可在下载资源处下载。
- **提交勘误**：您对书中内容的修改意见可在提交勘误处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方读者评论处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/33109

目 录 CONTENTS

CHAPTER

1

第 1 章
Spring Cloud 概述

本章要点

- 传统应用的问题
- 微服务与 Spring Cloud
- 学习方面的准备工作

本章将会简述 Spring Cloud 的功能，描述什么是 Spring Cloud，它能为我们带来什么，为后面学习该框架的知识打下理论基础。

1.1 传统的应用

1.1.1 单体应用

在此之前，笔者所在公司开发 Java 程序，大都使用 Struts、Spring、Hibernate（MyBatis）等技术框架，每一个项目都会发布一个单体应用。例如开发一个进销存系统，将会开发一个 war 包部署到 Tomcat 中，每一次需要开发新的模块或添加新功能时，都会在原来的基础上不断地添加。若干年后，这个 war 包不断膨胀，程序员在进行调试时，服务器也可能需要启动半天，维护这个系统的效率极为低下。这样一个 war 包，涵盖了库存、销售、会员、报表等模块，如图 1-1 所示。

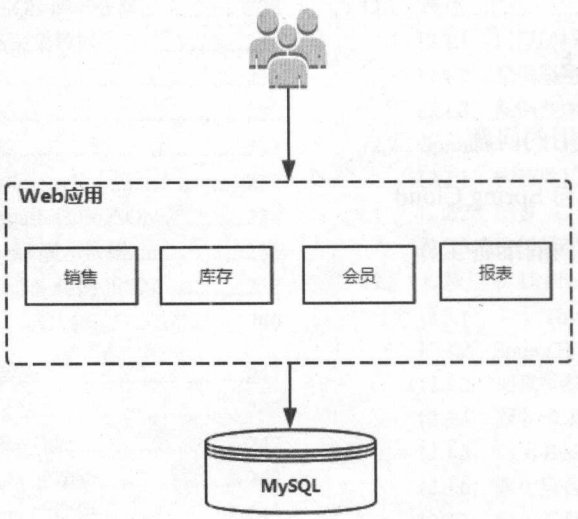


图 1-1 单体应用

这样的单体应用隐患非常多，任何一个 bug 都有可能导致整个系统宕机。笔者印象最深的是，曾经有一个客户在高峰期导出一张销售明细报表（数据量较大），最终造成整个系统瘫痪，前台的销售人员无法售卖。维护这样一个系统，不仅效率极低，而且充满风险，项目组的各个成员惶惶不可终日，我们需要本质上的改变。

1.1.2 架构演进

针对以上所说的单体应用的问题，我们参考 SOA 架构，将各个模块划分为独立的服务模块（war），并且使用了数据库的读写分离，架构如图 1-2 所示。

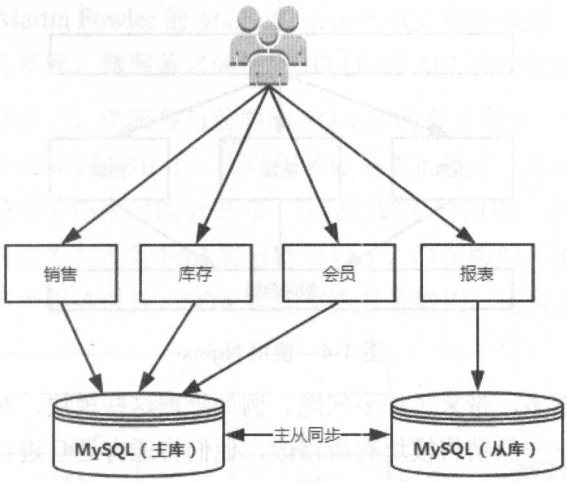


图 1-2 架构演进

各个模块之间会存在相互调用的依赖关系，例如销售模块会调用会员模块的接口，为了减少各个模块之间的耦合，我们加入了企业服务总线（ESB），各模块与 ESB 之间的架构如图 1-3 所示。

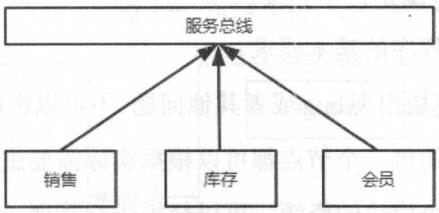


图 1-3 ESB

加入 ESB 后，各个模块将服务发布到 ESB 中，它们与 ESB 之间使用 SOAP 协议进行通信。图 1-2 与图 1-3 所示的架构实现后，整个系统的性能有了明显提升，各个模块的耦合度也降低了。运行了一段时间后，又出现了新的问题，由于销售终端数量的增多，销售模块明显超过其承受能力，为了保证销售前端的正常运行，我们使用了 Nginx 做负载均衡，请见图 1-4。

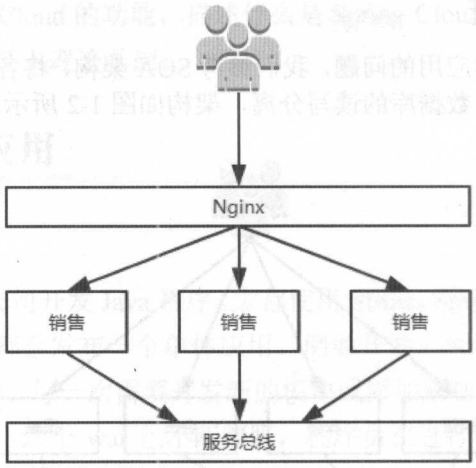


图 1-4 使用 Nginx

随着销售模块的增多，带来了许多问题。例如管理这些模块，对于运维工程师来说，这是一项艰巨的任务，一旦销售模块有所修改，他们将通宵达旦进行升级。另外，企业服务总线也有可能成为性能的瓶颈，虽然目前仍未出现该问题，但我们需要未雨绸缪。

1.1.3 架构要求

从前面的架构演进可知，应用中的每一个点都有可能成为系统的问题点。随着互联网应用的普及，在大数据、高并发的环境下，系统架构需要面对更为严苛的挑战，我们需要一套新的架构，它起码应能满足以下要求。

- **高性能**：这是应用程序的基本要求。
- **独立性**：其中一个模块出现bug或者其他问题，不可以影响其他模块或者整个应用。
- **容易扩展**：应用中的每一个节点都可以根据实际需要进行扩展。
- **便于管理**：对于各个模块的资源，可以轻松进行管理、升级，减少维护成本。
- **状态监控与警报**：对整个应用程序进行监控，当某一个节点出现问题时，能及时发出警报。

为了能解决遇到的问题、达到以上的架构要求，我们开始研究 Spring Cloud。

1.2 微服务与 Spring Cloud

1.2.1 什么是微服务

微服务一词来自 Martin Fowler 的 *Microservices* 一文，微服务是一种架构风格，将单体应用划分为小型的服务单元，微服务之间使用 HTTP 的 API 进行资源访问与操作。

在对单体应用的划分上，微服务与前面的 SOA 架构有点类似，但是 SOA 架构侧重于将每个单体应用的服务集成到 ESB 上，而微服务做得更加彻底，强调将整个模块变成服务组件，微服务对模块的划分粒度可能会更细。以我们前面的销售、会员模块为例，在 SOA 架构中，只需将相应的服务发布到 ESB 容器就可以了，而在微服务架构中，这两个模块本身，将会变为一个或多个服务组件。SOA 架构与微服务架构，请参见图 1-5 与图 1-6。

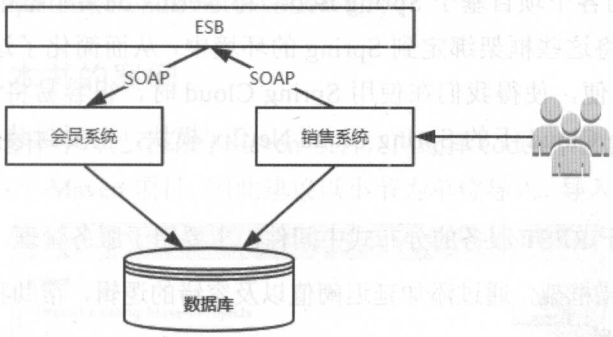


图 1-5 SOA 架构

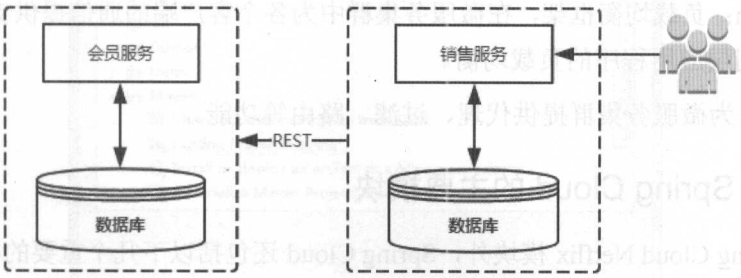


图 1-6 微服务架构

在微服务的架构上，Martin Fowler 的文章肯定了 Netflix 的贡献，接下来，我们了解一下 Netflix OSS。

1.2.2 关于 Netflix OSS

Netflix 是一个互联网影片提供商，在几年前，Netflix 公司成立了自己的开源中心，名称为 Netflix Open Source Software Center，简称 Netflix OSS。这个开源组织专注于大数据、云计算方面的技术，提供了多个开源框架，这些框架包括大数据工具、构建工具、基于云平台的服务工具等。Netflix 所提供的这些框架，很好地遵循了微服务所推崇的理念，实现了去中心化的服务管理、服务容错等机制。

1.2.3 Spring Cloud 与 Netflix

Spring Cloud 并不是一个具体的框架，大家可以把它理解为一个工具箱，它提供的各类工具，可以帮助我们快速构建分布式系统。

Spring Cloud 的各个项目基于 Spring Boot，将 Netflix 的多个框架进行封装，并且通过自动配置的方式将这些框架绑定到 Spring 的环境中，从而简化了这些框架的使用。由于 Spring Boot 的简便，使得我们在使用 Spring Cloud 时，很容易将 Netflix 各个框架整合进项目中。Spring Cloud 下的 Spring Cloud Netflix 模块，主要封装了 Netflix 的以下项目。

- Eureka：基于 REST 服务的分布式中间件，主要用于服务管理。
- Hystrix：容错框架，通过添加延迟阈值以及容错的逻辑，帮助我们控制分布式系统间组件的交互。
- Feign：一个 REST 客户端，目的是为了简化 Web Service 客户端的开发。
- Ribbon：负载均衡框架，在微服务集群中为各个客户端的通信提供支持，它主要实现中间层应用程序的负载均衡。
- Zuul：为微服务集群提供代理、过滤、路由等功能。

1.2.4 Spring Cloud 的主要模块

除了 Spring Cloud Netflix 模块外，Spring Cloud 还包括以下几个重要的模块。

- Spring Cloud Config：为分布式系统提供了配置服务器和配置客户端，通过对它们的配置，可以很好地管理集群中的配置文件。
- Spring Cloud Sleuth：服务跟踪框架，可以与 Zipkin、Apache HTrace 和 ELK 等数据分析、服务跟踪系统进行整合，为服务跟踪、解决问题提供了便利。

- Spring Cloud Stream：用于构建消息驱动微服务的框架，该框架在 Spring Boot 的基础上，整合了 Spring Integration 来连接消息代理中间件。
- Spring Cloud Bus：连接 RabbitMQ、Kafka 等消息代理的集群消息总线。

1.3 学习方面的准备工作

1.3.1 下载本书的软件及源码

读者可以到 www.broadview.com.cn 的“下载资源”处下载本书各章示例的源代码及涉及的一些软件。

读者也可发邮件与笔者联系，邮箱地址为 yangenxiong@163.com；也可以直接访问笔者博客直接交流，https://my.oschina.net/JavaLaw/blog。

1.3.2 导入本书的案例

在 Eclipse 中，选择导入已存在的 Maven 项目，再选择 codes 下的具体目录即可。由于每个案例都会包含多个 Maven 项目，因此建议以小节为单位导入，导入界面请参见图 1-7。

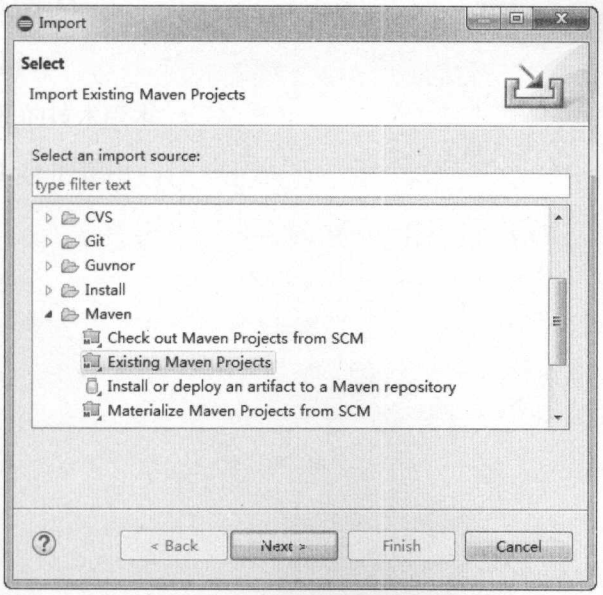


图 1-7 导入本书案例

在导入案例前，请先按照第 2 章的介绍把开发环境搭建好。

1.4 本章小结

本章的 1.1 节，对传统的单体应用、SOA 架构做了一个简单的总结，在此过程中分析了我们所遇到的问题。在 1.2 节，简单介绍了微服务与 Spring Cloud。接下来，我们正式开始讲述 Spring Cloud 的知识点。

CHAPTER

2

第2章 搭建开发环境

本章要点

- 安装与配置 Maven
- 安装 Eclipse
- 本书涉及的技术版本
- Spring Boot 的使用

工欲善其事，必先利其器。在讲述本书的技术内容前，先将开发环境搭建好，本书所涉及的基础环境将在本章准备，包括 Eclipse、Maven 等。如果读者对 Maven、Eclipse、Spring Boot 等项目较为熟悉，可以直接跳过本章的安装过程。

建议读者在学习本书的过程中，使用与本书相同的工具以及版本。本章使用的 Java 版本为 1.8，图 2-1 所示为“java –version”命令的输出，Java 安装与配置较为简单，本书不进行讲述。

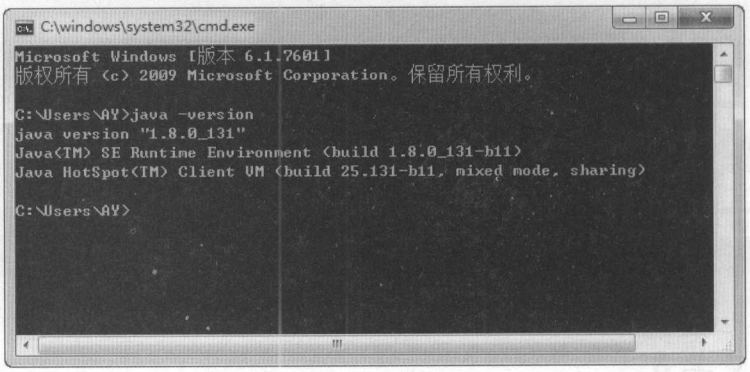


图 2-1 Java 版本

注意：

本书全部案例均在 Windows 7 下开发和运行。

2.1 安装与配置 Maven

2.1.1 关于 Maven

Maven 是 Apache 下的一个开源项目，用于项目的构建。使用 Maven 可以对项目的依赖包进行管理，支持构建脚本的继承。对于一些模块（子项目）较多的项目来说，Maven 是更好的选择，子项目可以继承父项目的构建脚本，减少了构建脚本的冗余。

除此之外，Maven 本身的插件机制让其更加强大和灵活，使用者可以配置各种 Maven 插件来完成不同的任务，如果感觉官方或者第三方提供的 Maven 插件不够用，还可以自行编写符合自己要求的 Maven 插件。Maven 为使用者提供了一个统一的依赖仓库，在上面可以找到各种开源项目的发布包，在一间公司或者一个项目组内部，甚至可以搭建私有的

Maven 仓库，将自己项目的包放到私有仓库中，供其他项目组或者开发者使用。

在 Maven 的众多特性中，最为重要的是它对依赖包的管理，Maven 将项目所使用的依赖包的信息放到 pom.xml 的 dependencies 节点中。例如需要使用 spring-core 模块的 jar 包，只需在 pom.xml 中配置该模块的依赖信息，Maven 会自动将 spring-beans 等模块引入项目的环境变量中。Spring Cloud 项目基于 Spring Boot 搭建，正是由于依赖管理的特性，使得 Maven 与 Spring Boot 更加相得益彰，可以让我们更快速地搭建一个可用的开发环境。

2.1.2 下载与安装 Maven

本书所使用的 Maven 的版本为 3.5，可以到 Maven 官方网站下载：http://maven.apache.org/。下载并解压后得到 apache-maven-3.5.0 目录，将主目录下的 bin 目录加入系统的环境变量中，如图 2-2 所示。

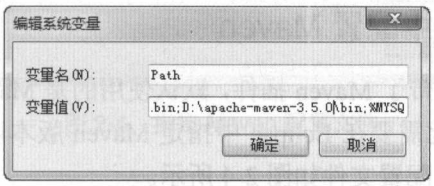


图 2-2 修改环境变量

配置完后，打开 cmd 命令行，输入“mvn–v”，可以看到输出的 Maven 版本信息。Maven 下载的依赖包会存放到本地仓库中，默认路径为：C:\Users\用户名\.m2\repository。

2.1.3 配置远程仓库

如果不进行仓库配置，默认情况下，Maven 会到 Apache 官方的仓库下载依赖包。由于 Apache 官方的仓库位于国外，下载速度较慢，会降低开发效率，笔者建议使用国内的 Maven 仓库或者搭建自己的私服，本书重点不是 Maven，因此直接使用了由阿里云提供的 Maven 仓库。修改 apache-maven-3.5.0/conf 目录下的 setting.xml，在 mirrors 节点下加入以下配置：

```
<mirror>
    <id>alimaven</id>
    <name>aliyun maven</name>
    <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
    <mirrorOf>central</mirrorOf>
</mirror>
```

配置完后，以后在使用过程中，Maven 会先到阿里云的仓库中下载依赖包。另外，需要注意的是，本书的大部分案例并没有使用 Maven 的继承特性，每一个 Maven 项目都可以独立引入。

2.2 安装 Eclipse

2.2.1 Eclipse 版本

本书使用 Eclipse 作为开发工具，使用的版本为 Luna（4.4），大家可以从以下地址得到该版本的 Eclipse：http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/lunasr2，也可以在本书所附代码的 soft 目录下找到该版本的 Eclipse。目前 Eclipse 已经发展到 4.7 版本，本书主要在 Eclipse 中使用 Maven 插件。

2.2.2 在 Eclipse 中配置 Maven

Luna 版本的 Eclipse 自带了 Maven 插件，默认使用的是 Maven 3.2。由于我们前面安装的是 Maven 3.5 版本，因此需要在 Eclipse 中指定 Maven 版本以及配置文件。指定 Maven 的配置如图 2-3 所示，指定配置文件如图 2-4 所示。

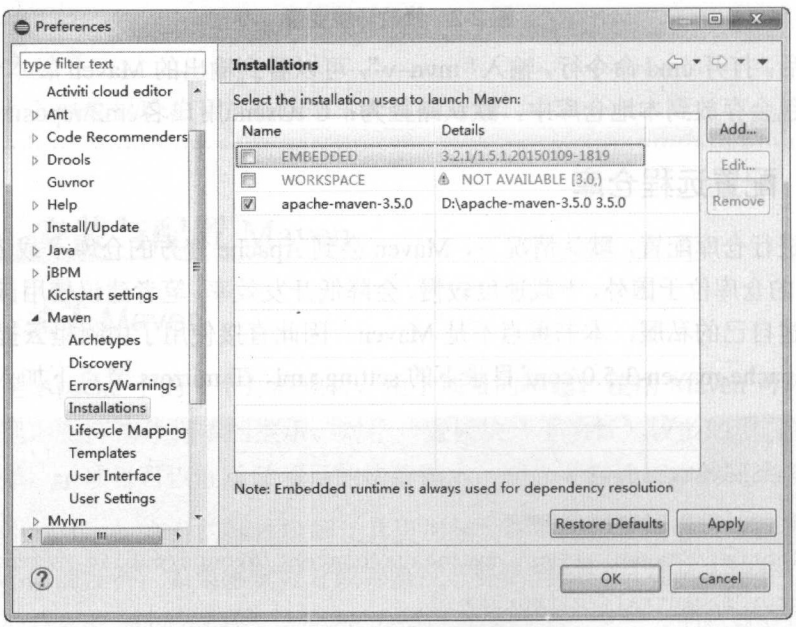


图 2-3 指定 Maven 版本

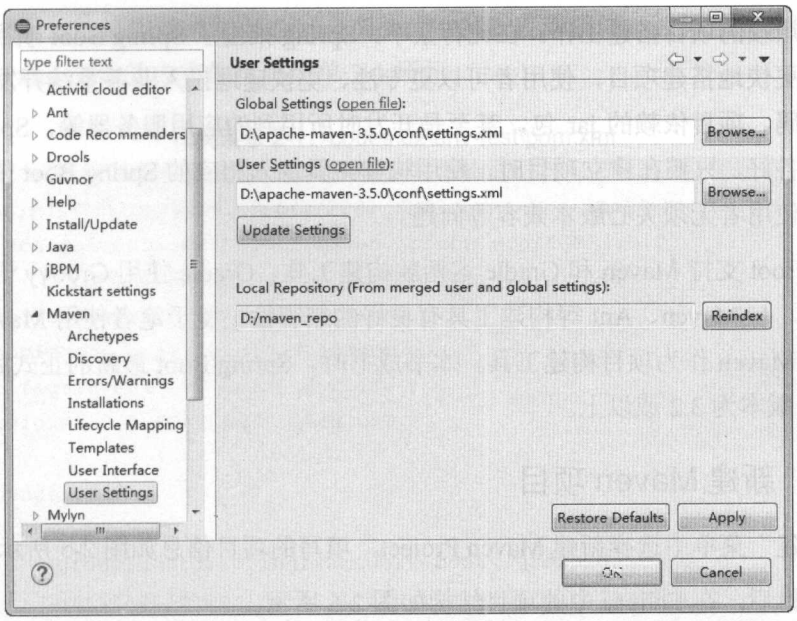


图 2-4 指定 Maven 配置文件

注意：

本书的案例如无特别说明均以 Maven 项目的形式导入。

2.3 Spring Boot

Spring Cloud 基于 Spring Boot 搭建，本节将对 Spring Boot 进行一个大致讲解，读者知道 Spring Boot 的作用即可。

2.3.1 Spring Boot 简介

开发一个全新的项目，需要先进行开发环境的搭建，例如要确定技术框架以及版本，还要考虑各个框架之间的版本兼容问题。完成这些烦琐的工作后，还要对新项目进行配置，测试能否正常运行，最后才能将搭建好的环境提交给项目组的其他成员使用。经常出现的情形是，表面上已经成功运行，但部分项目组成员仍然无法运行，项目初期浪费大量的时间做这些工作，几乎每个项目都会投入部分工作量来做这些固定的事情。

受 Ruby On Rails、Node.js 等技术的影响，Java EE 领域需要一种更为简便的开发方式

来取代这些烦琐的项目搭建工作。在此背景下，Spring 推出了 Spring Boot 项目，该项目可以让使用者更快地搭建项目，使用者可以更专注、更快速地投入业务系统开发中。系统配置、基础代码、项目依赖的 jar 包，甚至是开发时所用到的应用服务器等，Spring Boot 已经帮我们准备好，只要在建立项目时，使用构建工具加入相应的 Spring Boot 依赖包，项目即可运行，使用者无须关心版本兼容等问题。

Spring Boot 支持 Maven 和 Gradle 这两款构建工具。Gradle 使用 Groovy 语言进行构建脚本的编写，与 Maven、Ant 等构建工具有良好的兼容性。鉴于笔者使用 Maven 较多，因此本书使用 Maven 作为项目构建工具。本书成书时，Spring Boot 最新的正式版本为 1.5.4，要求 Maven 版本为 3.2 或以上。

2.3.2 新建 Maven 项目

在“新建”菜单中选择新建 Maven Project，填写的项目信息如图 2-5 所示。

新建项目后，在 Eclipse 中的项目结构如图 2-6 所示。

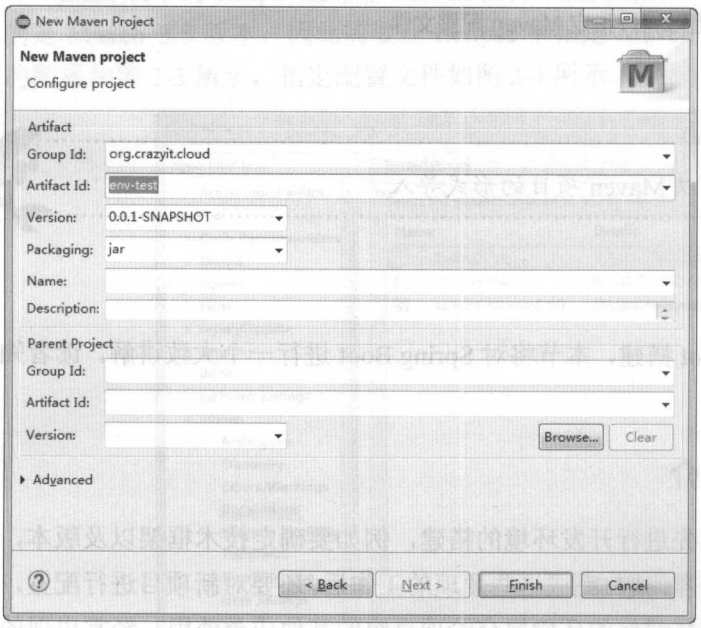


图 2-5 新建 Maven 项目

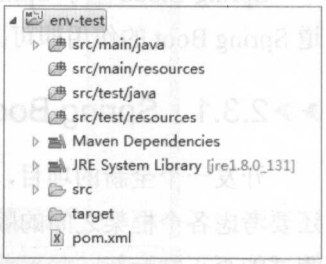


图 2-6 项目结构

这是一个最基本的 Maven 项目结构，在 src/main/java 中存放 Java 文件，src/main/resources 用于存放项目的资源文件，而 src/test 则用于存放测试相关的源代码及资源。

为了测试项目的可用性，加入 Spring Boot 的 Web 启动模块，让该项目具有 Web 容器的功能，pom.xml 文件的内容如代码清单 2-1 所示。

代码清单 2-1：codes\02\env-test\pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
          http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.crazyit.cloud</groupId>
    <artifactId>env-test</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <version>1.5.4.RELEASE</version>
        </dependency>
    </dependencies>
</project>
```

配置完依赖后，该依赖会自动帮我们的项目加上其他的 Spring 模块以及所依赖的第三方包，例如 spring-core、spring-beans、spring-mvc 等，除了这些模块外，还加入了嵌入式的 Tomcat。

2.3.3 编写启动类

加入了依赖后，只需要编写一个简单的启动类即可启动 Web 服务，启动类如代码清单 2-2 所示。

代码清单 2-2：codes\02\env-test\src\main\java\org\crazyit\cloud\MyApplication.java

```
@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

MyApplication 类使用了@SpringBootApplication 注解，该注解声明了该类是一个 Spring Boot 应用，该注解具有@SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan 等注解的功能。直接运行 MyApplication 的 main 方法，看到以下输出信息后，证明成功启动：

```
    2017-08-02 20:53:05.327  INFO 1976 --- [          main]
o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] onto
handler of type [class
org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
    2017-08-02 20:53:05.530  INFO 1976 --- [          main]
o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on
startup
    2017-08-02 20:53:05.878  INFO 1976 --- [          main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
    2017-08-02 20:53:05.885  INFO 1976 --- [          main]
org.crazyit.cloud.MyApplication          : Started MyApplication in 5.758 seconds
(JVM running for 6.426)
```

根据输出信息可知，启动的 Tomcat 端口为 8080，打开浏览器访问 http://localhost:8080，可以看到错误页面，表示应用已经成功启动。

2.3.4 编写控制器

在前面小节中加入的 spring-boot-starter-web 模块，默认集成了 Spring MVC，因此只需要编写一个 Controller，即可实现一个最简单的 HelloWorld 程序，代码清单 2-3 所示为控制器。

代码清单 2-3：codes\02\env-test\src\main\java\org\crazyit\cloud\MyController.java

```
@Controller
public class MyController {

    @GetMapping("/hello")
    @ResponseBody
    public String hello() {
        return "Hello World";
    }
}
```

代码清单 2-3 中使用了@Controller 注解来修饰 MyController，由于启动类中使用了@SpringBootApplication 注解，该注解含有@ComponentScan 的功能，因此@Controller 会被扫描并注册。在 hello 方法中使用了@GetMapping 与@ResponseBody 注解，声明 hello 方法的访问地址以及返回内容。重新运行启动类，打开浏览器并访问以下地址 http://localhost:8080/hello，可以看到控制器的返回。

2.3.5 发布 REST WebService

Spring MVC 支持直接发布 REST 风格的 WebService，新建测试的对象 Person，如代码清单 2-4 所示。

代码清单 2-4：codes\02\env-test\src\main\java\org\crazyit\cloud\Person.java

```
public class Person {

    private Integer id;

    private String name;

    private Integer age;

    ……省略 setter 和 getter 方法
}
```

修改控制器类，修改后如代码清单 2-5 所示。

代码清单 2-5：codes\02\env-test\src\main\java\org\crazyit\cloud\MyController.java

```
@RestController
public class MyController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello World";
    }

    @RequestMapping(value = "/person/{personId}", method = RequestMethod.GET,
        produces = MediaType.APPLICATION_JSON_VALUE)
    public Person findPerson(@PathVariable("personId") Integer personId) {
        Person p = new Person();
        p.setId(personId);
        p.setName("Crazyit");
        p.setAge(30);
        return p;
    }
}
```

在 MyController 类中，将原来的@Controller 注解修改为@RestController，原来的 hello 方法也不需要使用@ResponseBody 进行修饰，因为@RestController 中已含有@ResponseBody 注解。新建 findPerson 方法，该方法将会根据参数 id 来创建一个 Person 实例并返回，访问该

方法将会得到 JSON 字符串。运行启动类，在浏览器中输入 http://localhost:8080/person/1，可看到接口返回以下 JSON 字符串：

```
{"id":1,"name":"Crazyit","age":30}
```

调用 REST 服务的方式有很多，此部分内容将在后面章节中讲述。

2.4 Spring Boot 配置文件

Spring Cloud 基于 Spring Boot 构建，很多模块的配置均放在 Spring Boot 的配置文件中，因此有必要了解 Spring Boot 的配置文件规则，为后面的学习打下基础。

2.4.1 默认配置文件

Spring Boot 会按顺序读取各种配置，例如命令行参数、系统参数等，本章只讲述配置文件的参数读取。默认情况下，Spring Boot 会按顺序到以下目录读取 application.properties 或者 application.yml 文件：

- 项目根目录的 config 目录。
- 项目根目录。
- 项目 classpath 下的 config 目录。
- 项目 classpath 根目录。

如对以上描述有疑问，可参看图 2-7。

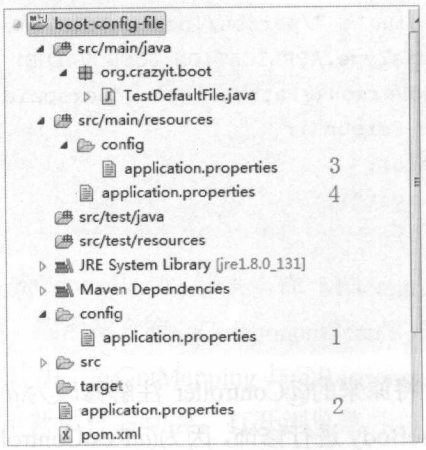


图 2-7 配置文件读取顺序

图 2-7 中所示的数字为文件的读取顺序，本小节使用的 boot-config-file 项目依赖了 spring-boot-starter-web 项目，为 pom.xml 加入以下依赖：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>1.5.4.RELEASE</version>
    </dependency>
</dependencies>
```

2.4.2 指定配置文件位置

如果想自己指定配置文件，可以在 Spring 容器的启动命令中加入参数，例子如代码清单 2-6 所示。

代码清单 2-6：codes\02\boot-config-file\src\main\java\org\crazyit\boot\TestDefaultFile.java

```
ConfigurableApplicationContext context =
    new SpringApplicationBuilder(TestDefaultFile.class)
    .properties(
        "spring.config.location=classpath:/test-folder/my-config.properties")
    .run(args);
```

在使用 SpringApplicationBuilder 时，TestDefaultFile 类配置了 spring.config.location 属性来设定需要读取的配置文件。

2.4.3 yml 文件

YAML 语言使用一种方便的格式进行数据配置，通过配置分层、缩进，在很大程度上增强了配置文件的可读性，使用 YAML 语言的配置文件以 yml 作为后缀。代码清单 2-7 为一份 yml 配置文件。

代码清单 2-7：codes\02\boot-config-file\src\main\resources\my-config.yml

```
jdbc:
  user:
    root
  passwd:
    123456
  driver:
    com.mysql.jdbc.Driver
```

在此，需要注意的是，每一行配置的缩进要使用空格，不要使用 Tab 键进行缩进。代码清单 2-7 对应的 properties 文件内容如下所示：

```
jdbc.user=root
jdbc.passwd=123456
jdbc.driver=com.mysql.jdbc.Driver
```

2.4.4 运行时指定 profiles 配置

如果在不同的环境下激活不同的配置，可以使用 profiles，代码清单 2-8 中配置了两个 profiles。

代码清单 2-8：codes\02\boot-config-file\src\main\resources\test-profiles.yml

```
spring:
  profiles: mysql
jdbc:
  driver:
    com.mysql.jdbc.Driver
---
spring:
  profiles: oracle
jdbc:
  driver:
    oracle.jdbc.driver.OracleDriver
```

定义了 mysql 与 oracle 两个 profiles，profiles 间使用“---”进行分隔。在 Spring 容器启动时，使用 spring.profiles.active 来指定激活哪个 profiles，如代码清单 2-9 所示。

代码清单 2-9：codes\02\boot-config-file\src\main\java\org\crazyit\boot\TestProfiles.java

```
ConfigurableApplicationContext context =
    new SpringApplicationBuilder(TestProfiles.class)
    .properties(
        "spring.config.location=classpath:/test-profiles.yml")
    .properties("spring.profiles.active=oracle")
    .run(args);
```

在对 Spring Boot 的配置文件有一定了解后，对后面章节中 Spring Cloud 的配置内容就不会陌生了。

2.4.5 热部署

每次修改 Java 文件后，都需要重新运行 main 方法才能生效，这样会降低开发效率，

我们可以使用 Spring Boot 提供的开发工具来实现热部署，为项目加上以下依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

当 Java 文件修改后，容器会重新加载本项目的 Java 类。

2.5 Spring Cloud 的版本

本书使用的 Spring Cloud 版本为 Dalston.SR1，相关的技术版本如下所示。

- spring-cloud-commons：公共模块，版本为 1.2.2.RELEASE。
- spring-cloud-config：配置管理模块，版本为 1.3.1.RELEASE。
- spring-cloud-netflix：Spring Cloud 的核心模块，用于提供服务管理、负载均衡等功能，版本为 1.3.1.RELEASE。
- spring-cloud-sleuth：服务跟踪模块，版本为 1.2.1.RELEASE。
- spring-cloud-stream：用于构建消息驱动微服务的模块，版本为 Chelsea.SR2。
- spring-cloud-bus：消息总线模块，对应版本为 1.3.1.RELEASE。
- spring-boot：Spring Cloud 基于 Spring Boot 快速搭建，使用的 Spring Boot 版本为 1.5.3.RELEASE。

2.6 本章小结

本章主要讲述了本书所需基础环境的搭建，读者需掌握 Maven 的使用，本书的案例几乎都是 Maven 项目。Spring Cloud 项目以 Spring Boot 作为基础进行构建，本书的大部分案例也是基于 Spring Boot 的。本章对 Spring Boot 进行了大致的讲解，并且配合一个 Hello World 例子来演示 Spring Boot 的便捷。学习完本章后，读者了解 Spring Boot 的大致功能即可。

CHAPTER

3

第3章 微服务发布与调用

本章要点

- 认识 Eureka 框架
- 运行 Eureka 服务器
- 发布微服务
- 调用微服务
- 服务器端配置
- 服务提供者配置
- 搭建 Eureka 集群

本章将讲述 Spring Cloud 中 Eureka 的使用，包括在 Eureka 服务器上发布、调用微服务，Eureka 的配置以及 Eureka 集群等内容。

3.1 Eureka 介绍

Spring Cloud 集成了 Netflix OSS 的多个项目，形成了 spring-cloud-netflix 项目。该项目包含多个子模块，这些子模块对集成的 Netflix 旗下的框架进行了封装，本节将讲述其中一个较为重要的服务管理框架：Eureka。

3.1.1 关于 Eureka

Eureka 提供基于 REST 的服务，在集群中主要用于服务管理。Eureka 提供了基于 Java 语言的客户端组件，客户端组件实现了负载均衡的功能，为业务组件的集群部署创造了条件。使用该框架，可以将业务组件注册到 Eureka 容器中，这些组件可进行集群部署，Eureka 主要维护这些服务的列表并自动检查它们的状态。

3.1.2 Eureka 架构

一个简单的 Eureka 集群，需要一个 Eureka 服务器、若干个服务提供者。我们可以将业务组件注册到 Eureka 服务器中，其他客户端组件可以向服务器获取服务并且进行远程调用。图 3-1 所示为 Eureka 的架构图。

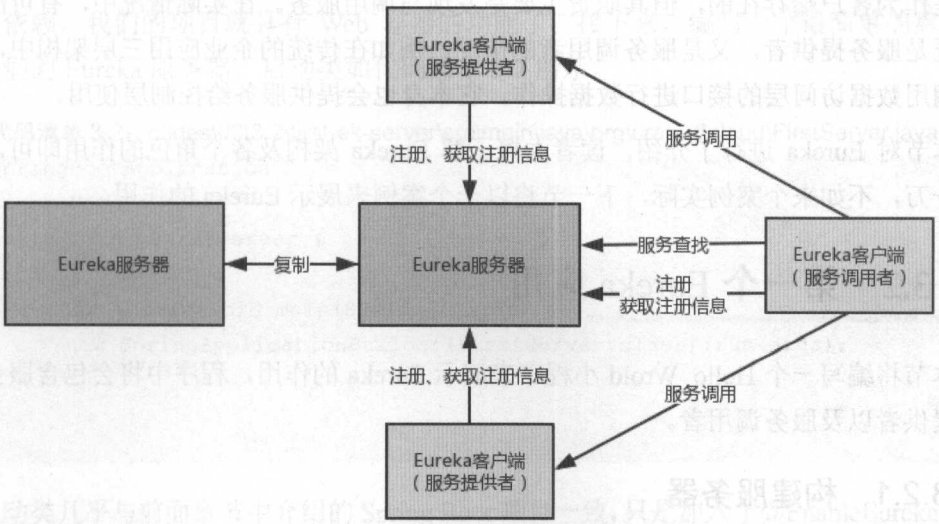


图 3-1 Eureka 架构图

图 3-1 中有两个服务器，服务器支持集群部署，每个服务器也可以作为对方服务器的客户端进行相互注册与复制。图 3-1 中所示的三个 Eureka 客户端，两个用于发布服务，另一个用于调用服务。不管是服务器还是客户端，都可以部署多个实例，如此一来，就很容易构建高可用的服务集群。

3.1.3 服务器端

对于注册到服务器端的服务组件，Eureka 服务器并没有提供后台的存储，这些注册的服务实例被保存在内存的注册中心，它们通过心跳来保持其最新状态，这些操作都可以在内存中完成。客户端存在着相同的机制，同样在内存中保存了注册表信息，这样的机制提升了 Eureka 组件的性能，每次服务的请求都不必经过服务器端的注册中心。

3.1.4 服务提供者

作为 Eureka 客户端存在的服务提供者，主要进行以下工作：第一，向服务器注册服务；第二，发送心跳给服务器；第三，向服务器端获取注册列表。当客户端注册到服务器时，它将会提供一些关于自己的信息给服务器端，例如自己的主机、端口、健康检测连接等。

3.1.5 服务调用者

对于发布到 Eureka 服务器的服务，服务调用者可对其进行服务查找与调用，服务调用者也是作为客户端存在的，但其职责主要是发现与调用服务。在实际情况中，有可能出现本身既是服务提供者，又是服务调用者的情况，例如在传统的企业应用三层架构中，服务层会调用数据访问层的接口进行数据操作，它本身也会提供服务给控制层使用。

本节对 Eureka 进行了介绍，读者大概了解 Eureka 架构及各个角色的作用即可，说一千道一万，不如来个案例实际，下一节将以一个案例来展示 Eureka 的作用。

3.2 第一个 Eureka 应用

本节将编写一个 Hello Wrold 小程序来演示 Eureka 的作用，程序中将会包含服务器、服务提供者以及服务调用者。

3.2.1 构建服务器

先创建一个名称为 first-ek-server 的 Maven 项目作为服务器，在 pom.xml 文件中加入

Spring Cloud 的依赖，如代码清单 3-1 所示。

代码清单 3-1：codes\03\3.2\first-ek-server\pom.xml

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Dalston.SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka-server</artifactId>
        </dependency>
    </dependencies>
```

加入的 spring-cloud-starter-eureka-server 会自动引入 spring-boot-starter-web，因此只需加入该依赖，我们的项目就具有 Web 容器的功能了。接下来，编写一个最简单的启动类，启动我们的 Eureka 服务器，启动类如代码清单 3-2 所示。

代码清单 3-2：codes\03\3.2\first-ek-server\src\main\java\org\crazyit\cloud\FirstServer.java

```
@SpringBootApplication
@EnableEurekaServer
public class FirstServer {

    public static void main(String[] args) {
        new SpringApplicationBuilder(FirstServer.class).run(args);
    }

}
```

启动类几乎与前面章节中介绍的 Spring Boot 项目一致，只是加入了@EnableEurekaServer，声明这是一个 Eureka 服务器。直接运行 FirstServer 即可启动 Eureka 服务器，需要注意的是，

本例中并没有配置服务器端口，因此默认端口为 8080，我们将端口配置为 8761，在 src/main/resources 目录下创建 application.yml 配置文件，内容如下：

```
server:
  port: 8761
```

本书的大部分案例使用 yml 文件进行配置，以上的配置片断声明服务器的 HTTP 端口为 8761，该配置是 Spring Boot 的公共配置。Spring Boot 有近千个公共配置，如想了解这些配置，可参看 Spring Boot 的文档。运行 FirstServer 的 main 方法后，可以看到控制台的输出如下：

```
    2017-08-04 15:35:58.900  INFO 4028 --- [          main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8761 (http)
    2017-08-04 15:35:58.901  INFO 4028 --- [          main]
.s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 8761
    2017-08-04 15:35:58.906  INFO 4028 --- [          main]
org.crazyit.cloud.FirstServer            : Started FirstServer in 12.361 seconds (JVM
running for 12.891)
    2017-08-04 15:35:59.488  INFO 4028 --- [nio-8761-exec-1]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring FrameworkServlet
'dispatcherServlet'
```

在启动过程中会出现部分异常信息，暂时不需要进行处理，将在后面章节讲解如何避免出现这些异常。成功启动后，打开浏览器，输入 http://localhost:8761，可以看到 Eureka 服务器控制台，如图 3-2 所示。

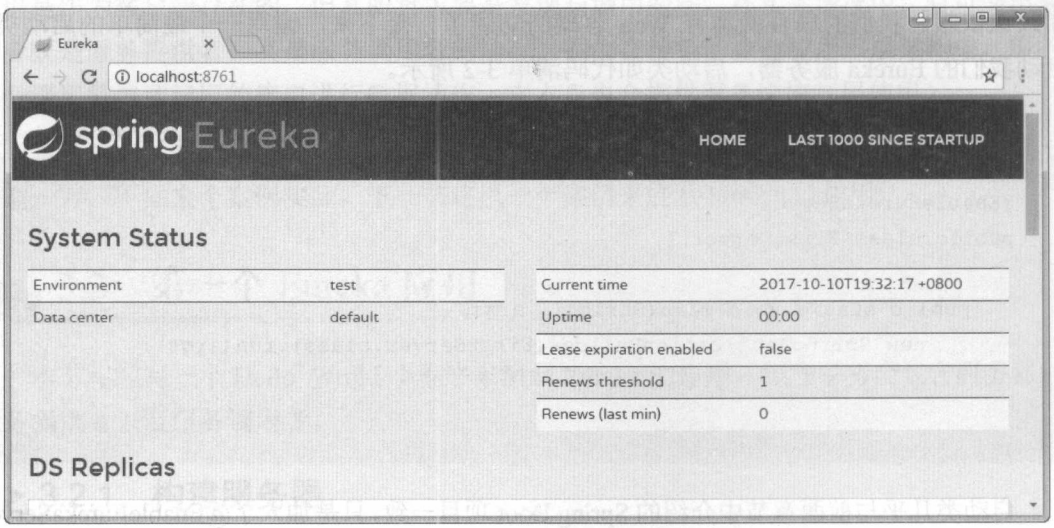


图 3-2 Eureka 控制台

在图 3-2 的下方，可以看到服务的实例列表，目前我们并没有注册服务，因此列表为空。

3.2.2 服务器注册开关

在启动 Eureka 服务器时，会在控制台看到以下两个异常信息：

```
    java.net.ConnectException: Connection refused: connect
    com.netflix.discovery.shared.transport.TransportException: Cannot execute
request on any known server
```

这是由于在服务器启动时，服务器会把自己当作一个客户端，去注册 Eureka 服务器，并且会到 Eureka 服务器抓取注册信息，它自己本身只是一个服务器，而不是服务的提供者（客户端），因此可以修改 application.yml 文件，修改以下两个配置：

```
    eureka:
      client:
        registerWithEureka: false
        fetchRegistry: false
```

以上配置中的 eureka.client.registerWithEureka 属性，声明是否将自己的信息注册到 Eureka 服务器，默认值为 true。属性 eureka.client.fetchRegistry 则表示，是否到 Eureka 服务器中抓取注册信息。将这两个属性设置为 false，启动时不会出现异常信息。

3.2.3 编写服务提供者

在前面搭建环境章节，我们使用 Spring Boot 来建立一个简单的 Web 工程，并且在里面编写了一个 REST 服务，本例中的服务提供者，与该案例类似。建立名称为 first-ek-service-provider 的项目，在 pom.xml 中加入依赖，如下所示。

```
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>
```

在 src/main/resources 目录中建立 application.yml 配置文件，文件内容如代码清单 3-3 所示。

代码清单 3-3：codes\03\3.2\first-ek-service-provider\src\main\resources\application.yml

```
spring:
  application:
    name: first-service-provider
eureka:
  instance:
    hostname: localhost
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

以上配置中，将应用名称配置为 first-service-provider，该服务将会被注册到端口为 8761 的 Eureka 服务器，也就是本节前面所构建的服务器。另外，还使用了 eureka.instance.hostname 来配置该服务实例的主机名称。编写一个 Controller 类，并提供一个最简单的 REST 服务，如代码清单 3-4 所示。

代码清单 3-4：codes\03\3.2\first-ek-service-provider\src\main\java\org\crazyit\cloud\FirstController.java

```
@RestController
public class FirstController {

    @RequestMapping(value = "/person/{personId}", method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON_VALUE)
    public Person findPerson(@PathVariable("personId") Integer personId) {
        Person person = new Person(personId, "Crazyit", 30);
        return person;
    }
}
```

编写启动类，参见代码清单 3-5。

代码清单 3-5：

codes\03\3.2\first-ek-service-provider\src\main\java\org\crazyit\cloud\FirstServiceProvider.java

```
@SpringBootApplication
@EnableEurekaClient
public class FirstServiceProvider {
```

```
    public static void main(String[] args) {
        new SpringApplicationBuilder(FirstServiceProvider.class).run(args);
    }
}
```

在启动类中，使用了@EnableEurekaClient 注解，声明该应用是一个 Eureka 客户端。配置完成后，运行服务器项目 first-ek-server 的启动类 FirstServer，再运行代码清单 3-5 中的 FirstServiceProvider，在浏览器中访问 Eureka：http://localhost:8761/，可以看到服务列表如图 3-3 所示。

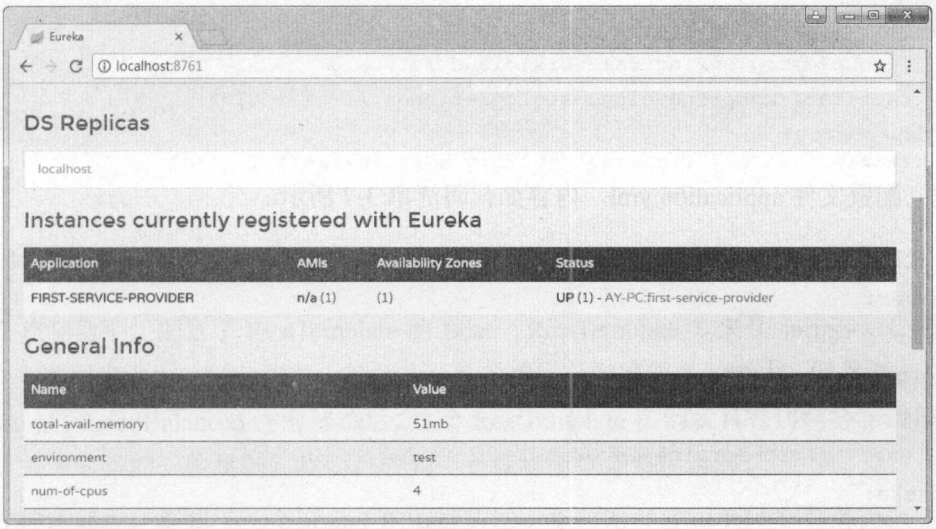


图 3-3 查看发布的服务

如图 3-3 所示，可以看到当前注册的服务列表，只有我们编写的 first-service-provider。服务注册成功后，接下来编写服务调用者。

3.2.4 编写服务调用者

服务被注册、发布到 Eureka 服务器后，需要有程序去发现它，并且进行调用。此处所说的调用者，是指同样注册到 Eureka 的客户端，来调用其他客户端发布的服务。简单地说，就是 Eureka 内部调用。同一个服务可能会部署多个实例，调用过程可能涉及负载均衡、服务器查找等问题，Netflix 的项目已经帮我们解决，并且 Spring Cloud 已经封装了一次，我们仅需编写少量代码就可以实现服务调用。

新建名称为 first-ek-service-invoker 的项目，在 pom.xml 文件中加入依赖，如代码清

单 3-6 所示。

代码清单 3-6：codes\03\3.2\first-ek-service-invoker\pom.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-ribbon</artifactId>
</dependency>
```

建立配置文件 application.yml，内容如代码清单 3-7 所示。

代码清单 3-7：codes\03\3.2\first-ek-service-invoker\src\main\resources\application.yml

```
server:
  port: 9000
spring:
  application:
    name: first-service-invoker
eureka:
  instance:
    hostname: localhost
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

在配置文件中，配置了应用名称为 first-service-invoker，这个调用者的访问端口为 9000，需要注意的是，这个调用本身也可以对外提供服务。与提供者一样，使用 eureka 的配置，将调用者注册到 first-ek-server 上面。下面编写一个控制器，让调用者对外提供一个测试的服务，代码清单 3-8 为控制器的代码实现。

代码清单 3-8：

codes\03\3.2\first-ek-service-invoker\src\main\java\org\crazyit\cloud\InvokerController.java

```
@RestController
```

```
@Configuration
public class InvokerController {

    @Bean
    @LoadBalanced
    public RestTemplate getRestTemplate() {
        return new RestTemplate();
    }

    @RequestMapping(value = "/router", method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON_VALUE)
    public String router() {
        RestTemplate restTpl = getRestTemplate();
        // 根据应用名称调用服务
        String json = restTpl.getForObject(
                "http://first-service-provider/person/1", String.class);
        return json;
    }
}
```

在控制器中，配置了 RestTemplate 的 Bean，RestTemplate 本来是 spring-web 模块下面的类，主要用来调用 REST 服务。本身并不具备调用分布式服务的能力，但是 RestTemplate 的 Bean 被@LoadBalanced 注解修饰后，这个 RestTemplate 实例就具有访问分布式服务的能力了。关于该类的一些机制，我们将放到“负载均衡”章节中讲解。

在控制器中，新建了一个 router 的测试方法，用来对外发布 REST 服务。该方法只起路由作用，实际上是使用 RestTemplate 来调用 first-ek-service-provider（服务提供者）的服务。需要注意的是，调用服务时，仅仅通过服务名称进行调用。接下来编写启动类，如代码清单 3-9 所示。

代码清单 3-9：codes\03\3.2\first-ek-service-invoker\src\main\java\org\crazyit\cloud\FirstInvoker.java

```
@SpringBootApplication
@EnableDiscoveryClient
public class FirstInvoker {

    public static void main(String[] args) {
        SpringApplication.run(FirstInvoker.class, args);
    }
}
```

在启动类中，使用了@EnableDiscoveryClient 注解来修改启动类，该注解使得服务调用者有能力去 Eureka 中发现服务。需要注意的是，@EnableEurekaClient 注解已经包含了@EnableDiscoveryClient 的功能，也就是说，一个 Eureka 客户端，本身就具有发现服务的能力。配置完成后，依次执行以下操作：

1. 启动服务器（first-ek-server）。
2. 启动服务提供者（first-ek-service-provider）。
3. 启动服务调用者（first-ek-service-invoker）。

使用浏览器访问 Eureka，可看到注册的客户信息，如图 3-4 所示。

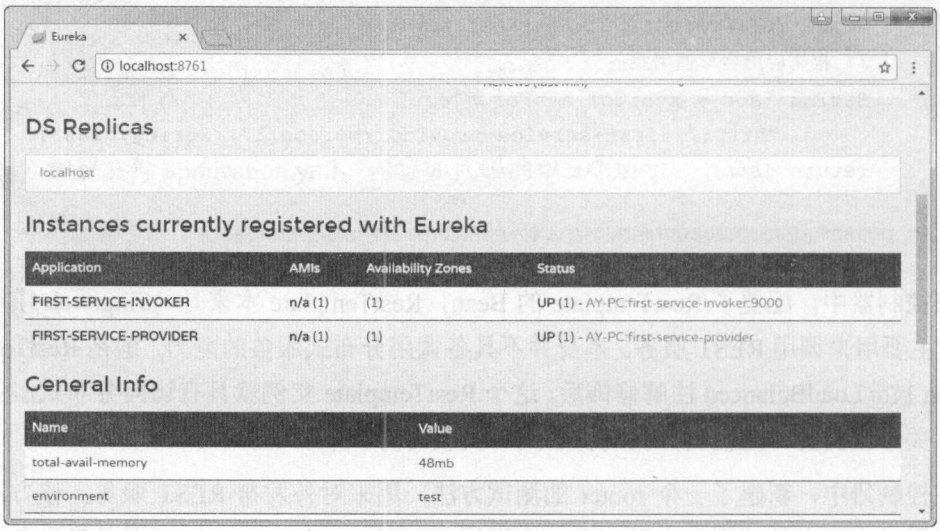


图 3-4 服务列表

全部成功启动后，在浏览器中访问服务调用者发布的 router 服务：http://localhost:9000/router，可以看到在浏览器中输出如下：

```
{"id":1,"name":"Crazyit","age":30}
```

根据输出可知，实际上调用了服务提供者的/person/1 服务，第一个 Eureka 应用到此结束，下面对这个应用程序的结构进行简单描述。

3.2.5 程序结构

本案例新建了三个项目，如果读者对程序的结构不太清晰，可以参看图 3-5。

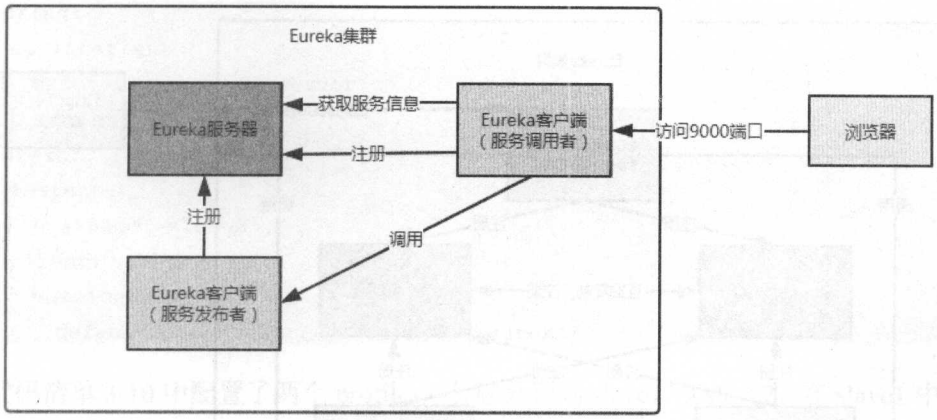


图 3-5 本案例的结构

如图 3-5 所示，Eureka 服务为本例的 first-ek-server，服务提供者为 first-ek-service-provider，而调用者为 first-ek-service-invoker，用户通过浏览器访问调用者的 9000 端口的 router 服务、router 服务中查找服务提供者的服务并进行调用。在本例中，服务调用有点像路由器的角色。

为了能演示 Eureka 的高可用特性，下一节将会以本案例为基础，搭建一个复杂一点的集群。

3.3 Eureka 集群搭建

在运行第一个 Eureka 应用时，服务器实例、服务提供者实例都只启动了一个，并没有体现高可用的特性，本节将对前面的 Eureka 应用进行改造，使其可以进行集群部署。

3.3.1 本例集群结构图

本例将会运行两个服务器实例、两个服务提供者实例，然后服务调用者请求服务，集群结构如图 3-6 所示。

第一个 Eureka 应用，使用的是浏览器访问 Eureka 的服务调用者，而改造后，为了能看到负载均衡的效果，会编写一个 HttpClient 的 REST 客户端访问服务调用者发布的服务。

由于本书的开发环境只有一台电脑，操作系统为 Windows，如果要构建集群，需要修改 hosts 文件，为其添加主机名的映射。修改 C:\Windows\System32\drivers\etc\hosts 文件，添加以下内容：

```
127.0.0.1 slave1 slave2
```

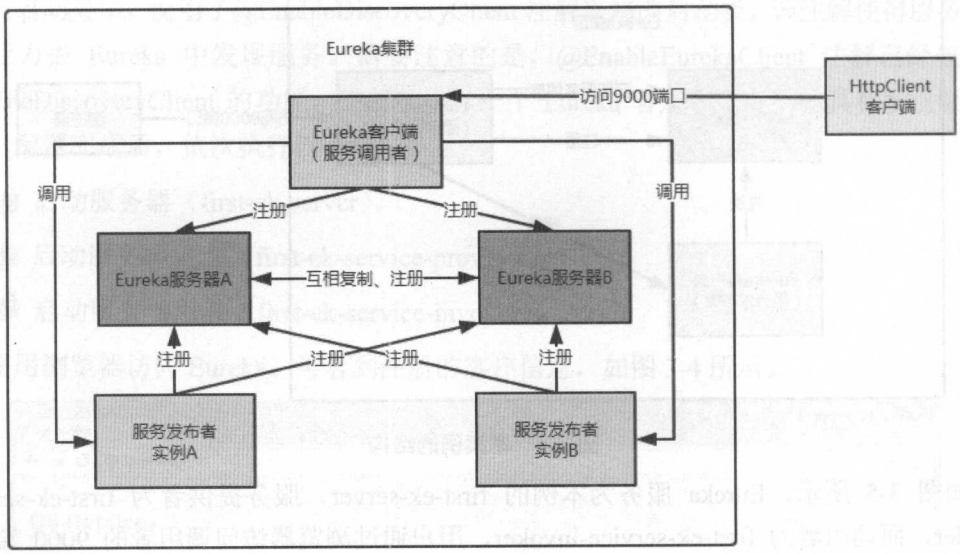


图 3-6　集群结构

➤➤ 3.3.2　改造服务器端

新建项目 first-cloud-server，使用的 Maven 配置与 3.2 节中介绍的服务器一致。由于需要对同一个应用程序启动两次，因此需要在配置文件中使用 profiles（关于 profiles 已经在第 2 章中讲述过）。服务器配置文件请见代码清单 3-10。

代码清单 3-10：codes\03\3.3\first-cloud-server\src\main\resources\application.yml

```
server:
  port: 8761
spring:
  application:
    name: first-cloud-server
  profiles: slave1
eureka:
  instance:
    hostname: slave1
  client:
    serviceUrl:
      defaultZone: http://slave2:8762/eureka/
---
server:
  port: 8762
```

```
spring:
  application:
    name: first-cloud-server
  profiles: slave2
eureka:
  instance:
    hostname: slave2
  client:
    serviceUrl:
      defaultZone: http://slave1:8761/eureka/
```

代码清单 3-10 中配置了两个 profiles，名称分别为 slave1 和 slave2。在 slave1 中，配置了应用端口为 8761，主机名为 slave1。当使用 salve1 这个 profiles 来启动服务器时，将会向 http://slave2:8762/eureka/注册自己。使用 salve2 来启动服务器，会向 http://slave1:8761/eureka/注册自己。**简单点说，就是两个服务器启动后，它们会互相注册。**

修改启动类，让类在启动时读取控制台的输入，决定使用哪个 profiles 来启动服务器，请见代码清单 3-11。

代码清单 3-11：codes\03\3.3\first-cloud-server\src\main\java\org\crazyit\cloud\FirstServer.java

```
@SpringBootApplication
@EnableEurekaServer
public class FirstServer {

    public static void main(String[] args) {
        // 读取控制台输入，决定使用哪个profiles
        Scanner scan = new Scanner(System.in);
        String profiles = scan.nextLine();
        new SpringApplicationBuilder(FirstServer.class)
                .profiles(profiles).run(args);
    }
}
```

在启动类中，先读取控制的输入，再调用 profiles 方法设置启动的 profiles。需要注意的是，第一个启动的服务器会抛出异常，异常原因我们在前面已经讲述过，抛出的异常不必理会。

3.3.3 改造服务提供者

服务提供者也需要启动两个实例，服务提供者的改造与服务端类似，将 3.2 节中的

first-ek-service-provider 复制出来，并改名为 first-cloud-provider。修改配置文件，将服务提供者注册到两个服务器中，配置文件请见代码清单 3-12。

代码清单 3-12：codes\03\3.3\first-cloud-provider\src\main\resources\application.yml

```
spring:
  application:
    name: first-cloud-provider
eureka:
  instance:
    hostname: localhost
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/,http://localhost:8762/eureka/
```

再修改启动类，为了避免端口冲突，启动时读取控制台输出，决定使用哪个端口来启动，启动类如代码清单 3-13 所示。

代码清单 3-13：

codes\03\3.3\first-cloud-provider\src\main\java\org\crazyit\cloud\FirstServiceProvider.java

```
// 读取控制台输入的端口，避免端口冲突
Scanner scan = new Scanner(System.in);
String port = scan.nextLine();
new SpringApplicationBuilder(FirstServiceProvider.class).properties(
    "server.port=" + port).run(args);
```

启动类中使用了 properties 方法来设置启动端口。为了能看到效果，还需要改造控制器，将服务调用者请求的 URL 保存起来并返回，修改后的控制器请见代码清单 3-14。

代码清单 3-14：codes\03\3.3\first-cloud-provider\src\main\java\org\crazyit\cloud\FirstController.java

```
@RestController
public class FirstController {

    @RequestMapping(value = "/person/{personId}", method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON_VALUE)
    public Person findPerson(@PathVariable("personId") Integer personId,
        HttpServletRequest request) {
        Person person = new Person(personId, "Crazyit", 30);
        // 为了查看结果，将请求的 URL 设置到 Person 实例中
        person.setMessage(request.getRequestURL().toString());
        return person;
    }
}
```

控制器的 findPerson 方法将请求的 URL 保存到 Person 实例的 message 属性中，调用服务后，可以通过 message 属性来查看请求的 URL。

3.3.4 改造服务调用者

将 3.2 节中的 first-ek-service-invoker 复制并改名为 first-cloud-invoker。本例中的服务调用者只需启动一个实例，因此修改配置文件即可使用，请见代码清单 3-15。

代码清单 3-15：codes\03\3.3\first-cloud-invoker\src\main\resources\application.yml

```
server:
  port: 9000
spring:
  application:
    name: first-cloud-invoker
eureka:
  instance:
    hostname: localhost
  client:
    serviceUrl:
      defaultZone: http://slave1:8761/eureka/,http://slave12:8762/eureka/
```

修改的配置将服务调用注册到两个服务器上。

3.3.5 编写 REST 客户端进行测试

本例使用的是 HttpClient，HttpClient 是 Apache 提供的一个 HTTP 工具包。新建名称为 first-cloud-rest-client 的项目，在 pom.xml 中加入以下依赖：

```
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.5.2</version>
</dependency>
```

新建启动类，在 main 方法中编写调用 REST 服务的代码，如代码清单 3-16 所示。

代码清单 3-16：codes\03\3.3\first-cloud-rest-client\src\main\java\org\crazyit\cloud\TestHttpClient.java

```
// 创建默认的 HttpClient
CloseableHttpClient httpclient = HttpClients.createDefault();
// 调用 6 次服务并输出结果
for(int i = 0; i < 6; i++) {
```

```
        // 调用 GET 方法请求服务
        HttpGet httpget = new HttpGet("http://localhost:9000/router");
        // 获取响应
        HttpResponse response = httpclient.execute(httpget);
        // 根据响应解析出字符串
        System.out.println(EntityUtils.toString(response.getEntity()));
    }
```

在 main 方法中，调用了 6 次 9000 端口的 router 服务并输出结果。完成编写后，按以下顺序启动各个组件：

1. 启动两个服务器端，在控制台分别输入 slave1 和 slave2。
2. 启动两个服务提供者，在控制台分别输入 8081 与 8082。
3. 启动服务调用者。

启动了整个集群后，运行 TestHttpClient，可以看到输出如下：

```
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8081/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8082/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8081/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8082/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8081/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8082/person/1"}
```

根据输出结果可知，8081 与 8082 端口分别被请求了 3 次，可见已经达到负载均衡的目的，关于负载均衡更详细的内容，将在后面章节中讲解。

3.4 服务实例的健康自检

在默认情况下，Eureka 的客户端每隔 30 秒会发送一次心跳给服务器端，告知它仍然存活。但是，在实际环境中有可能出现这种情况，客户端表面上可以正常发送心跳，但实际上服务是不可用的。

例如一个需要访问数据的服务提供者，表面上可以正常响应，但是数据库已经无法访问；又如，服务提供者需要访问第三方的服务，而这些服务早已失效。对于这些情况，应当告诉服务器当前客户的状态，调用者或者其他客户端无法获取这些有问题的实例。实现该功能，可以使用 Eureka 的健康检查控制器。

3.4.1 程序结构

将3.2节的服务器、服务提供者和服务调用者进行复制，命名如下。

- health-handler-server：本例的Eureka服务器。
- health-handler-provider：本例的服务提供者客户端。
- health-handler-invoker：本例的服务调用者客户端。

假设在实际环境中，服务提供者模块需要访问数据库，**本例的health-handler-provider模块，将是进行健康自检的模块。**

3.4.2 使用Spring Boot Actuator

Spring Boot Actuator模块主要用于系统监控，当应用程序整合了Actuator后，它就会自动提供多个服务端点，这些端点可以让外部看到应用程序的健康情况。在本例中使用的是/health端点。修改health-handler-provider的pom.xml文件，加入以下依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
    <version>1.5.3.RELEASE</version>
</dependency>
```

加入依赖后，先启动health-handler-server，再启动health-handler-provider。这两个模块与3.2节中介绍的基本类似，仅仅修改了部分类名。启动完成后，在浏览器中访问http://localhost:8080/health，可以看到输出如下：

```
{"description":"Spring Cloud Eureka Discovery Client","status":"UP"}
```

该REST服务向外展示当前应用的状态为“UP”。

3.4.3 实现应用健康自检

如果一个客户端本身没有问题，但是该模块所依赖的服务无法使用，那么对于服务器以及其他客户端来说，该客户端也是不可用的，最常见的就是访问数据库的模块。我们需要做两件事：第一，让客户端自己进行检查，是否能连接数据库；第二，将连接数据库的结果与客户端的状态进行关联，并且将状态告诉服务器。

我们使用Spring Boot Actuator可以直接实现一个自定义的HealthIndicator，根据是否能

访问数据库，来决定应用自身的健康。代码清单 3-17，为服务提供者（health-handler- provider）添加了一个健康指示器。

代码清单 3-17：

codes\03\3.4\health-handler-provider\src\main\java\org\crazyit\cloud\MyHealthIndicator.java

```
@Component
public class MyHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        if(HealthController.canVisitDb) {
            // 成功连接数据库，返回 UP
            return new Health.Builder(Status.UP).build();
        } else {
            // 连接数据库失败，返回 out of service
            return new Health.Builder(Status.DOWN).build();
        }
    }
}
```

为了简单起见，使用 HealthController 类的 canVisitDb 变量来模拟是否能连接上数据库，该变量的相关代码请见代码清单 3-18。

代码清单 3-18：

codes\03\3.4\health-handler-provider\src\main\java\org\crazyit\cloud\HealthController.java

```
@RestController
public class HealthController {

    // 标识当前数据库是否可以访问
    static Boolean canVisitDb = false;

    @RequestMapping(value = "/db/{canVisitDb}", method = RequestMethod.GET)
    public String setConnectState(@PathVariable("canVisitDb") Boolean canVisitDb) {
        this.canVisitDb = canVisitDb;
        return "当前数据库是否正常：" + this.canVisitDb;
    }
}
```

控制器中的 canVisitDb 变量，可以通过/db/false 或者/db/true 两个地址来修改，配合健

康指示器一起使用。如果该值为true，健康指示器将会返回“UP”状态，反之则返回“DOWN”状态。修改完后，启动服务器（health-handler-server）以及服务提供者（health-handler-provider），访问 http://localhost:8080/health 可以看到服务提供者的健康状态，默认状态为 DOWN，因此控制器中的 canVisitDb 默认值为 false。如果想让应用的健康状态变为 UP，则访问 http://localhost:8080/db/true 即可。

接下来，如果服务提供者想把健康状态告诉服务器，还需要实现“健康检查处理器”。处理器会将应用的健康状态保存到内存中，状态一旦发生改变，就会重新向服务器进行注册，其他的客户端将拿不到这些不可用的实例。代码清单 3-19 所示的是本例的健康检查处理器。

代码清单 3-19：

codes\03\3.4\health-handler-provider\src\main\java\org\crazyit\cloud\MyHealthCheckHandler.java

```
@Component
public class MyHealthCheckHandler implements HealthCheckHandler {

    @Autowired
    private MyHealthIndicator indicator;

    public InstanceStatus getStatus(InstanceStatus currentStatus) {
        Status s = indicator.health().getStatus();
        if(s.equals(Status.UP)) {
            System.out.println("数据库正常连接");
            return InstanceStatus.UP;
        } else {
            System.out.println("数据库无法连接");
            return InstanceStatus.DOWN;
        }
    }
}
```

在自定义的健康检查处理器中，注入了前面编写的健康指示器，根据健康指示器的结果来返回不同的状态。Eureka 中会启动一个定时器，定时刷新本地实例的信息，并且执行“处理器”中的 getStatus 方法，再将服务实例的状态“更新”到服务器中。执行以上逻辑的定时器，默认 30 秒执行一次，如果想加快看到效果，可以修改 eureka.client.instanceInfo-ReplicationIntervalSeconds 配置。代码清单 3-20 为服务器提供者的配置文件。

代码清单 3-20：codes\03\3.4\health-handler-provider\src\main\resources\application.yml

```
spring:
  application:
    name: health-handler-provider
eureka:
  instance:
    hostname: localhost
  client:
    instanceInfoReplicationIntervalSeconds: 10
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

启动服务器，再启动服务提供者，在浏览器中访问 http://localhost:8761，可看到服务提供者的状态为 DOWN，访问 http://localhost:8080/db/true，将数据库设置为“可以连接”，再访问 8761 端口，可以看到服务提供者状态已经改变为 UP。

3.4.4 服务查询

在前一节中，通过浏览器访问 Eureka 界面可查看服务状态的改变。本例通过修改服务调用者的代码来查看应用健康自检的效果。修改服务调用者（health-handler-invoker 模块），控制器修改后如代码清单 3-21 所示。

代码清单 3-21：

codes\03\3.4\health-handler-invoker\src\main\java\org\crazyit\cloud\InvokerController.java

```
@RestController
@Configuration
public class InvokerController {

    @Autowired
    private DiscoveryClient discoveryClient;

    @RequestMapping(value = "/router", method = RequestMethod.GET)
    public String router() {
        // 查找服务列表
        List<ServiceInstance> ins = getServiceInstances();
        // 输出服务信息及状态
        for (ServiceInstance service : ins) {
            EurekaServiceInstance esi = (EurekaServiceInstance) service;
            InstanceInfo info = esi.getInstanceInfo();
```

```
            System.out.println(info.getAppName() + "---" + info.getInstanceId()
                    + "---" + info.getStatus());
        }
        return "";
    }

    /**
     * 查询可用服务
     */
    private List<ServiceInstance> getServiceInstances() {
        List<String> ids = discoveryClient.getServices();
        List<ServiceInstance> result = new ArrayList<ServiceInstance>();
        for (String id : ids) {
            List<ServiceInstance> ins = discoveryClient.getInstances(id);
            result.addAll(ins);
        }
        return result;
    }
}
```

在客户端中，如果需要查询集群中的服务，可以使用 Spring Cloud 的 discoveryClient 类，或者 Eureka 的 eurekaClient 类，Spring Cloud 对 Eureka 进行了封装。本例中调用了 discoveryClient 的方法来查询服务实例(如代码清单 3-21 中的粗体代码)。在控制器的 router 方法中，仅将查询到的服务实例进行输出。修改完后，依次进行以下操作：

1. 运行服务器。
2. 服务提供者。
3. 服务调用者。
4. 在浏览器中输入 http://localhost:8080/db/true，将数据库设置为可以连接。
5. 在浏览器中输入 http://localhost:9000/router，控制台输出如下：
   - HEALTH-HANDLER-PROVIDER---AY-PC:health-handler-provider---UP
   - HEALTH-HANDLER-INVOKER---AY-PC:health-handler-invoker:9000---UP
6. 在浏览器中输入 http://localhost:8080/db/false，将数据库设置为不可连接。
7. 在浏览器中输入 http://localhost:9000/router，控制台输出如下：

- HEALTH-HANDLER-INVOKER---AY-PC:health-handler-invoker:9000---UP

根据输出结果可知，将数据库设置为不可连接后，可用的服务只剩下调用者自己，服务提供者已经不存在于服务列表中。

运行案例需要注意，默认情况下，客户端到服务器端抓取注册表会有一定的时间间隔，因此在设置数据库是否可以连接后，访问“调用者”查看效果时需要稍等一会儿。

3.5 Eureka 的常用配置

本节将讲述部分 Eureka 的常用配置。

3.5.1 心跳检测配置

客户端的实例会向服务器发送周期性的心跳，默认是 30 秒发送一次，可以通过修改客户端的 eureka.instance.leaseRenewalIntervalInSeconds 属性来改变这个时间。

服务器端接收心跳请求，如果在一定期限内没有接收到服务实例的心跳，那么会将该实例从注册表中清理掉，其他的客户端将会无法访问这个实例。这个期限默认值为 90 秒，可以通过修改客户端的 eureka.instance.leaseExpirationDurationInSeconds 属性来改变这个值。也就是说，服务器 90 秒没有收到客户端的心跳，就会将这个实例从列表中清理掉。但需要注意的是，清理注册表有一个定时器在执行，默认是 60 秒执行一次，如果将 leaseExpirationDurationInSeconds 设置为小于 60 秒，虽然符合删除实例的条件，但是还没到 60 秒，这个实例将仍然存在注册表中（因为还没有执行清理）。我们可以在服务器端配置 eureka.server.eviction-interval-timer-in-ms 属性来修改注册表的清理间隔，该属性的单位是毫秒。

需要特别注意，如果开启了自我保护模式，则实例不会被剔除。在测试时，为避免受自我保护模式的影响，建议先关闭自我保护模式，在服务器中配置：eureka.server.enable-self-preservation=false。

3.5.2 注册表抓取间隔

在默认情况下，客户端每隔 30 秒去服务器端抓取注册表（可用的服务列表），并且将服务器端的注册表保存到本地缓存中。可以通过修改 eureka.client.registryFetch-IntervalSeconds 配置来改变注册表抓取间隔，但仍然需要考虑性能，改为哪个值比较合适，

需要在性能与实时性方面进行权衡。

3.5.3 配置与使用元数据

框架自带的元数据，包括实例 id、主机名称、IP 地址等，如果需要自定义元数据并提供给其他客户端使用，可以配置 eureka.instance.metadata-map 属性来指定。元数据都会保存在服务器的注册表中，并且使用简单的方式与客户端进行共享。在正常情况下，自定义元数据不会改变客户端的行为，除非客户端知道这些元数据的含义，以下配置片断使用了元数据。

```
eureka:
  instance:
    hostname: localhost
    metadata-map:
      company-name: crazyit
```

配置了一个名为 company-name 的元数据，值为 crazyit，使用元数据的一方，可以调用 discoveryClient 的方法获取元数据，如以下代码所示：

```
@Autowired
private DiscoveryClient discoveryClient;

@RequestMapping(value = "/router", method = RequestMethod.GET)
public String router() {
    // 查询服务实例
    List<ServiceInstance> ins = discoveryClient.getInstances ("heart-beat-client");
    // 遍历实例并输出元数据值
    for(ServiceInstance service : ins) {
        System.out.println(service.getMetadata().get("company-name"));
    }
    return "";
}
```

使用方式较为简单，在此不再赘述。

3.5.4 自我保护模式

在开发过程中，经常可以在 Eureka 的主界面中看到红色字体的提醒，内容如下：

```
    EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT.
RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST
TO BE SAFE.
```

出现该提示意味着 Eureka 进入了自我保护模式。根据前面章节的介绍可知，客户端会定时发送心跳给服务器端，如果心跳的失败率超过一定比例，服务会将这些实例保护起来，并不会马上将其从注册表中剔除。此时对于另外的客户端来说，有可能会拿到一些无法使用的实例，这种情况可能会导致灾难的“蔓延”，这些情况可以使用容错机制予以解决，关于集群的容错机制，将在后面的章节中讲述。在开发过程中，为服务器配置 eureka.server.enable-self-preservation 属性，将值设置为 false 来关闭自我保护机制。关闭后再打开 Eureka 主界面，可以看到以下提示信息：

```
    THE SELF PRESERVATION MODE IS TURNED OFF.THIS MAY NOT PROTECT INSTANCE EXPIRY
IN CASE OF NETWORK/OTHER PROBLEMS.
```

自我保护模式已经关闭，在出现网络或者其他问题时，将不会保护过期的实例。

3.6 本章小结

本章讲述了 Eureka 框架，读者学习完本章后，可以掌握 Eureka 的架构、如何使用 Eureka 搭建集群等内容，最基本的是，能对 Spring Cloud 等技术有一个初步认识。

如果想更进一步，可以学习编写服务实例的自检程序。实际环境中的服务不可避免地依赖第三方环境，例如数据库、第三方服务等，因此，如果掌握了对程序的自检，可以使得我们的应用程序更加健壮。

CHAPTER

4

第4章 负载均衡

本章要点

- 认识 Ribbon
- 第一个 Ribbon 程序
- Ribbon 的负载均衡机制
- 在 Spring Cloud 中使用 Ribbon
- RestTemplate 的负载均衡机制

负载均衡是分布式架构的重点，负载均衡机制决定着整个服务集群的性能与稳定。根据前面章节的介绍可知，Eureka 服务实例可以进行集群部署，每个实例都均衡处理服务请求，那么这些请求是如何被分摊到各个服务实例中的呢？本章将讲解 Netflix 的负载均衡项目 Ribbon。

4.1 Ribbon 介绍

4.1.1 Ribbon 简介

Ribbon 是 Netflix 下的负载均衡项目，它在集群中为各个客户端的通信提供了支持，它主要实现中间层应用程序的负载均衡。Ribbon 提供以下特性：

- 负载均衡器，可支持插拔式的负载均衡规则。
- 对多种协议提供支持，例如 HTTP、TCP、UDP。
- 集成了负载均衡功能的客户端。

同为 Netflix 项目，Ribbon 可以与 Eureka 整合使用，Ribbon 同样被集成到 Spring Cloud 中，作为 spring-cloud-netflix 项目中的子模块。Spring Cloud 将 Ribbon 的 API 进行了封装，使用者可以使用封装后的 API 来实现负载均衡，也可以直接使用 Ribbon 的原生 API。

4.1.2 Ribbon 子模块

Ribbon 主要有以下三大子模块。

- ribbon-core：该模块为 Ribbon 项目的核心，主要包括负载均衡器接口定义、客户端接口定义、内置的负载均衡实现等 API。
- ribbon-eureka：为 Eureka 客户端提供的负载均衡实现类。
- ribbon-httpclient：对 Apache 的 HttpClient 进行封装，该模块提供了含有负载均衡功能的 REST 客户端。

4.1.3 负载均衡器组件

Ribbon 的负载均衡器主要与集群中的各个服务器进行通信，负载均衡器需要提供以下基础功能：

- 维护服务器的 IP、DNS 名称等信息。

- 根据特定的逻辑在服务器列表中循环。

为了实现负载均衡的基础功能，Ribbon 的负载均衡器有以下三大子模块。

- Rule：一个逻辑组件，这些逻辑将会决定从服务器列表中返回哪个服务器实例。
- Ping：该组件主要使用定时器来确保服务器网络可以连接。
- ServerList：服务器列表，可以通过静态的配置确定负载的服务器，也可以动态指定服务器列表。如果动态指定服务器列表，则会有后台的线程来刷新该列表。

本章关于 Ribbon 的知识，主要围绕负载均衡器组件进行。接下来，我们先编写一个 Ribbon 程序，让大家对 Ribbon 有一个初步的认识。

4.2 第一个 Ribbon 程序

本章的 4.2 节和 4.3 节，单独使用 Ribbon 框架，关于整合 Spring Cloud 的内容，将在 4.4 节讲述。本节将以一个简单的 Hello World 程序来展示 Ribbon API 的使用。本例的程序结构如图 4-1 所示。

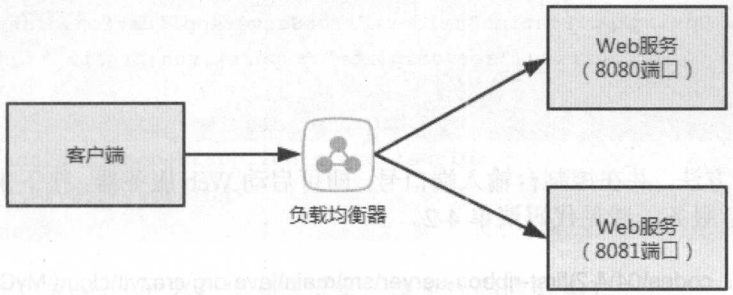


图 4-1 本例的程序结构

本书所使用的 Spring Cloud，默认集成的 Ribbon 版本为 2.2.2，因此本书也使用该版本的 Ribbon。

4.2.1 编写服务

为了能查看负载均衡效果，先编写一个简单的 REST 服务，通过指定不同的端口，让服务可以启动多个实例。本例的请求服务器，仅仅是一个基于 Spring Boot 的 Web 应用，与 2.3 节中的应用类似，如果读者熟悉建立过程，可跳过部分创建过程，本小节最终目的是发布两个 REST 服务。

新建名称为 first-ribbon-server 的 Maven 项目，加入以下依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>1.5.4.RELEASE</version>
</dependency>
```

建立 Spring Boot 启动类，如代码清单 4-1 所示。

代码清单 4-1：

codes\04\4.2\first-ribbon-server\src\main\java\org\crazyit\cloud\FirstServerApplication.java

```
@SpringBootApplication
public class FirstServerApplication {

    public static void main(String[] args) {
        // 读取控制台输入作为端口参数
        Scanner scan = new Scanner(System.in);
        String port = scan.nextLine();
        // 设置启动的服务器端口
        new SpringApplicationBuilder(FirstServerApplication.class)
                .properties("server.port=" + port).run(args);
    }
}
```

运行 main 方法，并在控制台输入端口号，即可启动 Web 服务器。接下来编写控制器，添加一个 REST 服务，请见代码清单 4-2。

代码清单 4-2：codes\04\4.2\first-ribbon-server\src\main\java\org\crazyit\cloud\MyController.java

```
@RestController
public class MyController {

    @RequestMapping(value = "/person/{personId}", method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON_VALUE)
    public Person findPerson(@PathVariable("personId") Integer personId,
            HttpServletRequest request) {
        Person p = new Person();
        p.setId(personId);
        p.setName("Crazyit");
        p.setAge(30);
        p.setMessage(request.getRequestURL().toString());
```

```
        return p;
    }

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String hello() {
        return "hello";
    }
}
```

在控制器中，发布了两个 REST 服务。其中，调用地址为/person/personId 的服务后，会返回一个 Person 实例的 JSON 字符串，为了看到请求的 URL，为 Person 的 message 属性设置了请求的 URL。

4.2.2 编写请求客户端

新建名称为 first-ribbon-client 的 Maven 项目，加入以下依赖：

```
<dependency>
    <groupId>com.netflix.ribbon</groupId>
    <artifactId>ribbon</artifactId>
    <version>2.2.2</version>
</dependency>
<dependency>
    <groupId>com.netflix.ribbon</groupId>
    <artifactId>ribbon-httpclient</artifactId>
    <version>2.2.2</version>
</dependency>
```

接下来，使用 Ribbon 的客户端发送请求，请见代码清单 4-3。

代码清单 4-3：codes\04\4.2\first-ribbon-client\src\main\java\org\crazyit\cloud\TestRestClient.java

```
public class TestRestClient {

    public static void main(String[] args) throws Exception {
        // 设置请求的服务器
        ConfigurationManager.getConfigInstance().setProperty(
                "my-client.ribbon.listOfServers",
                "localhost:8080,localhost:8081");
        // 获取 REST 请求客户端
        RestClient client = (RestClient) ClientFactory
```

```
                .getNamedClient("my-client");
        // 创建请求实例
        HttpRequest request = HttpRequest
            .newBuilder()
            .uri("/person/1").build();
        // 发送 6 次请求到服务器中
        for (int i = 0; i < 6; i++) {
            HttpResponse response = client.executeWithLoadBalancer(request);
            String result = response.getEntity(String.class);
            System.out.println(result);
        }
    }
}
```

在代码清单 4-3 中，使用 ConfigurationManager 类来配置请求的服务器列表，为 localhost:8080 与 localhost:8081，再使用 RestClient 对象，向/person/1 地址发送 6 次请求。

启动两次服务器类 FirstServerApplication，并在控制台分别输入 8080 和 8081 端口。启动服务器后，运行客户端，输出结果如下：

```
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8081/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8080/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8081/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8080/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8081/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8080/person/1"}
```

根据输出结果可知，RestClient 轮流向 8080 与 8081 端口发送请求，可见在 RestClient 中已经帮我们实现了负载均衡的功能。

4.2.3 Ribbon 的配置

在编写客户端时，使用了 ConfigurationManager 来设置配置项，除了在代码中指定配置项外，还可以将配置放到.properties 文件中。ConfigurationManager 的 loadPropertiesFromResources 方法可以指定 properties 文件的位置，配置格式如下：

```
<client>.<nameSpace>.<property>=<value>
```

其中<client>为客户的名称，声明该配置属于哪一个客户端，在使用 ClientFactory 时可传入客户端的名称，即可返回对应的“请求客户端”实例。<nameSpace>为该配置的命名

空间，默认为 ribbon，<property>为属性名，<value>为属性值。如果想对全部客户端生效，可以将客户端名称去掉，直接以<namespace>.<property>的格式进行配置。以下的配置为客户端指定了服务器列表：

```
my-client.ribbon.listOfServers=localhost:8080,localhost:8081
```

Ribbon 的配置同样可以在 Spring Cloud 的配置文件（即 application.yml）中使用。

4.3 Ribbon 的负载均衡机制

Ribbon 提供了几个负载均衡的组件，其目的就是让请求转给合适的服务器处理。因此，如何选择合适的服务器便成为负载均衡机制的核心。本节将围绕 Ribbon 负载均衡器的组件，向大家展示 Ribbon 负载均衡的实现机制。

4.3.1 负载均衡器

Ribbon 的负载均衡器接口定义了服务器的操作，主要是用于进行服务器选择。在前面的例子中，客户端使用了 RestClient 类，在发送请求时，会使用负载均衡器（ILoadBalancer）接口，根据特定的逻辑来选择服务器。服务器列表可使用 listOfServers 进行配置，也可以使用动态更新机制。代码清单 4-4 使用负载均衡器来选择服务器。

代码清单 4-4：codes\04\4.2\first-ribbon-client\src\main\java\org\crazyit\cloud\ChoseServerTest.java

```
// 创建负载均衡器
ILoadBalancer lb = new BaseLoadBalancer();
// 添加服务器
List<Server> servers = new ArrayList<Server>();
servers.add(new Server("localhost", 8080));
servers.add(new Server("localhost", 8081));
lb.addServers(servers);
// 进行 6 次服务器选择
for(int i = 0; i < 6; i++) {
    Server s = lb.chooseServer(null);
    System.out.println(s);
}
```

代码中使用了 BaseLoadBalancer 这个负载均衡器，将两个服务器对象加入负载均衡器中，再调用 6 次 chooseServer 方法，可以看到输出如下：

```
localhost:8081
localhost:8080
localhost:8081
localhost:8080
localhost:8081
localhost:8080
```

根据结果可知，最终选择的服务器与4.2节中介绍的一致，可以判定本例与4.2节的例子选择服务器的逻辑是一致的，在默认情况下，会使用RoundRobinRule的规则逻辑。

4.3.2 自定义负载规则

根据前一小节的介绍可知，选择哪个服务器进行请求处理，由ILoadBalancer接口的chooseServer方法决定。而在BaseLoadBalancer类中，则使用IRule接口的choose方法来决定选择哪一个服务器对象。如果想自定义负载均衡规定，可以编写一个IRule接口的实现类。代码清单4-5实现了自己的负载规定。

代码清单4-5：codes\04\4.2\first-ribbon-client\src\main\java\org\crazyit\cloud\MyRule.java

```
public class MyRule implements IRule {

    ILoadBalancer lb;

    public MyRule() {
    }

    public MyRule(ILoadBalancer lb) {
        this.lb = lb;
    }

    public Server choose(Object key) {
        // 获取全部的服务器
        List<Server> servers = lb.getAllServers();
        // 只返回第一个 Server 对象
        return servers.get(0);
    }

    public void setLoadBalancer(ILoadBalancer lb) {
        this.lb = lb;
    }
```

```
    public ILoadBalancer getLoadBalancer() {
        return this.lb;
    }
}
```

在自定义规则类中，实现的 choose 方法调用了 ILoadBalancer 的 getAllServers 方法，返回全部服务器，为了简单起见，本例只返回第一个服务器。为了能在负载均衡器中使用自定义的规则，需要修改选择服务器的代码，请见代码清单 4-6。

代码清单 4-6：codes\04\4.2\first-ribbon-client\src\main\java\org\crazyit\cloud\TestMyRule.java

```
// 创建负载均衡器
BaseLoadBalancer lb = new BaseLoadBalancer();
// 设置自定义的负载规则
lb.setRule(new MyRule(lb));
// 添加服务器
List<Server> servers = new ArrayList<Server>();
servers.add(new Server("localhost", 8080));
servers.add(new Server("localhost", 8081));
lb.addServers(servers);
// 进行 6 次服务器选择
for(int i = 0; i < 6; i++) {
    Server s = lb.chooseServer(null);
    System.out.println(s);
}
```

运行代码清单 4-6 可以看到，请求 6 次所得到的服务器均为 localhost:8080。以上是直接使用编码方式来设置负载规则，可以使用配置的方式来完成这些工作。修改 Ribbon 的配置，让请求的客户端使用我们定义的负载规则，请见代码清单 4-7。

代码清单 4-7：codes\04\4.2\first-ribbon-client\src\main\java\org\crazyit\cloud\TestMyRuleConfig.java

```
// 设置请求的服务器
ConfigurationManager.getConfigInstance().setProperty(
    "my-client.ribbon.listOfServers",
    "localhost:8080,localhost:8081");
// 配置规则处理类
ConfigurationManager.getConfigInstance().setProperty(
    "my-client.ribbon.NFLoadBalancerRuleClassName",
    MyRule.class.getName());
// 获取 REST 请求客户端
RestClient client = (RestClient) ClientFactory.
```

```
        .getNamedClient("my-client");
    // 创建请求实例
    HttpRequest request = HttpRequest.newBuilder().uri("/person/1").build();
    // 发送 6 次请求到服务器中
    for (int i = 0; i < 6; i++) {
        HttpResponse response = client.executeWithLoadBalancer(request);
        String result = response.getEntity(String.class);
        System.out.println(result);
    }
```

在请求客户端时，与 4.2 节中介绍的客户端基本一致，只是加入了 my-client.ribbon.NFLoadBalancerRuleClassName 属性，设置了自定义规则处理类为 MyRule，这个配置项同样可以在配置文件中使用，包括 Spring Cloud 的配置文件（application.yml 等）。

启动 4.2 节中介绍的服务器端两次，分别设置 8080 与 8081 端口，再运行代码清单 4-7，可以看到输出了 6 次{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8080/person/1"}，根据结果可知，我们的自定义规则生效了，请求只让 8080 端口处理。

在实际环境中，如果要实现自定义的负载规则，可能还需要结合各种因素，例如考虑具体业务的发生时间、服务器性能等。实现中可能还涉及使用计算器、数据库等技术，具体情形会更为复杂，本例的负载规则较为简单，目的是让读者了解负载均衡的原理。

4.3.3 Ribbon 自带的负载规则

Ribbon 提供了若干个内置的负载规则，使用者完全可以直接使用，主要有以下内置的负载规则。

- RoundRobinRule：系统默认的规则，通过简单地轮询服务列表来选择服务器，其他规则在很多情况下仍然使用 RoundRobinRule。
- AvailabilityFilteringRule：该规则会忽略以下服务器。
  - 无法连接的服务器：在默认情况下，如果 3 次连接失败，该服务器将会被置为“短路”的状态，该状态将持续 30 秒；如果再次连接失败，“短路”状态的持续时间将会以几何级数增加。可以通过修改 niws.loadbalancer.<clientName>.connectionFailureCountThreshold 属性，来配置连接失败的次数。
  - 并发数过高的服务器：如果连接到该服务器的并发数过高，也会被这个规则忽略，可以通过修改<clientName>.ribbon.ActiveConnectionsLimit 属性来设定最高并发数。

- WeightedResponseTimeRule：为每个服务器赋予一个权重值，服务器的响应时间越长，该权重值就越少，这个规则会随机选择服务器，权重值有可能会决定服务器的选择。
- ZoneAvoidanceRule：该规则以区域、可用服务器为基础进行服务器选择。使用 Zone 对服务器进行分类，可以理解为机架或者机房。
- BestAvailableRule：忽略“短路”的服务器，并选择并发数较低的服务器。
- RandomRule：顾名思义，随机选择可用的服务器。
- RetryRule：含有重试的选择逻辑，如果使用 RoundRobinRule 选择的服务器无法连接，那么将会重新选择服务器。

以上提供的负载规则基本可以满足大部分的需求，如果有更为复杂的要求，建议实现自定义负载规则。

4.3.4 Ping 机制

在负载均衡器中，提供了 Ping 机制，每隔一段时间，会去 Ping 服务器，判断服务器是否存活。该工作由 IPing 接口的实现类负责，如果单独使用 Ribbon，在默认情况下，不会激活 Ping 机制，默认的实现类为 DummyPing。代码清单 4-8 使用了另外一个 IPing 实现类 PingUrl。

代码清单 4-8：codes\04\4.2\first-ribbon-client\src\main\java\org\crazyit\cloud\TestPingUrl.java

```
// 创建负载均衡器
BaseLoadBalancer lb = new BaseLoadBalancer();
// 添加服务器
List<Server> servers = new ArrayList<Server>();
// 8080 端口连接正常
servers.add(new Server("localhost", 8080));
// 一个不存在的端口
servers.add(new Server("localhost", 8888));
lb.addServers(servers);
// 设置 IPing 实现类
lb.setPing(new PingUrl());
// 设置 Ping 时间间隔为 2 秒
lb.setPingInterval(2);
Thread.sleep(6000);
```

```
for(Server s : lb.getAllServers()) {
    System.out.println(s.getHostPort() + " 状态: " + s.isAlive());
}
```

代码清单 4-8 使用了代码的方法来设置负载均衡器使用 PingUrl，设置了每隔 2 秒就向两个服务器发起请求，PingUrl 实际使用的是 HttpClient。在以上例子中，实际上会请求 http://localhost:8080 与 http://localhost:8888 这两个地址，在运行前先以 8080 端口启动 4.2 节中介绍的服务器，最终效果为 8080 的服务器状态正常，而 8888 的服务器则无法连接，运行代码清单 4-8，可以看到输出如下：

```
localhost:8080 状态: true
localhost:8888 状态: false
```

除了在代码中配置使用 IPing 类外，还可以在配置中设置 IPing 实现类，请见代码清单 4-9。

代码清单 4-9：codes\04\4.2\first-ribbon-client\src\main\java\org\crazyit\cloud\TestPingUrlConfig.java

```
// 设置请求的服务器
ConfigurationManager.getConfigInstance().setProperty(
    "my-client.ribbon.listOfServers",
    "localhost:8080,localhost:8888");
// 配置 Ping 处理类
ConfigurationManager.getConfigInstance().setProperty(
    "my-client.ribbon.NFLoadBalancerPingClassName",
    PingUrl.class.getName());
// 配置 Ping 时间间隔
ConfigurationManager.getConfigInstance().setProperty(
    "my-client.ribbon.NFLoadBalancerPingInterval",
    2);
// 获取 REST 请求客户端
RestClient client = (RestClient) ClientFactory
    .getNamedClient("my-client");
Thread.sleep(6000);
// 获取全部服务器
List<Server> servers = client.getLoadBalancer().getAllServers();
System.out.println(servers.size());
// 输出状态
```

```
for(Server s : servers) {
    System.out.println(s.getHostPort() + " 状态：" + s.isAlive());
}
```

注意代码中的以下两个配置。

- my-client.ribbon.NFLoadBalancerPingClassName：配置 IPing 的实现类。
- my-client.ribbon.NFLoadBalancerPingInterval：配置 Ping 操作的时间间隔。

以上两个配置同样可以使用在配置文件中。

4.3.5 自定义 Ping

通过前面章节的案例可知，实现自定义 Ping 较为简单，先实现 IPing 接口，然后再通过配置来设定具体的 Ping 实现类，代码清单 4-10 为自定义的 Ping 类。

代码清单 4-10：codes\04\4.2\first-ribbon-client\src\main\java\org\crazyit\cloud\MyPing.java

```
public class MyPing implements IPing {

    public boolean isAlive(Server server) {
        System.out.println("这是自定义 Ping 实现类：" + server.getHostPort());
        return true;
    }
}
```

要使用自定义的 Ping 类，通过修改<client>.<nameSpace>.NFLoadBalancerPingClassName 配置即可，在此不再赘述。

4.3.6 其他配置

本节主要介绍了 Ribbon 负载均衡器的负载规则以及 Ping，这两部分可以通过配置来实现逻辑的改变。除了这两部分外，还可以使用以下配置来改变负载均衡器的其他行为。

- NFLoadBalancerClassName：指定负载均衡器的实现类，可利用该配置实现自己的负载均衡器。
- NIWSServerListClassName：服务器列表处理类，用来维护服务器列表，Ribbon 已经实现动态服务器列表。
- NIWSServerListFilterClassName：用于处理服务器列表拦截。

4.4 在 Spring Cloud 中使用 Ribbon

Spring Cloud 集成了 Ribbon，结合 Eureka，可实现客户端的负载均衡。前面章节中所使用的 RestTemplate（被@LoadBalanced 修饰）、还有后面章节中将介绍的 Feign，都已经拥有负载均衡功能。本节将以 RestTemplate 为基础，讲述及测试 Eureka 中的 Ribbon 配置。

4.4.1 准备工作

为了本节的测试做准备，按顺序进行以下工作：

- 新建 Eureka 服务器端项目，命名为 cloud-server，端口为 8761，代码目录为 codes\04\4.4\cloud-server。
- 新建 Eureka 服务提供者项目，命名为 cloud-provider，代码目录为 codes\04\4.4\cloud-provider，该项目主要进行以下工作。
  - 在控制器里面发布一个 REST 服务，地址为/person/{personId}，请求后返回 Person 实例，其中 Person 的 message 为 HTTP 请求的 URL。
  - 服务提供者需要启动两次，因此在控制台中需要输入启动端口。
- 新建 Eureka 服务调用者项目，命名为 cloud-invoker，对外端口为 9000，代码目录为 codes\04\4.4\cloud-invoker。本例的负载均衡配置主要针对服务调用者。

以上项目准备完成并启动后，结构如图 4-2 所示。

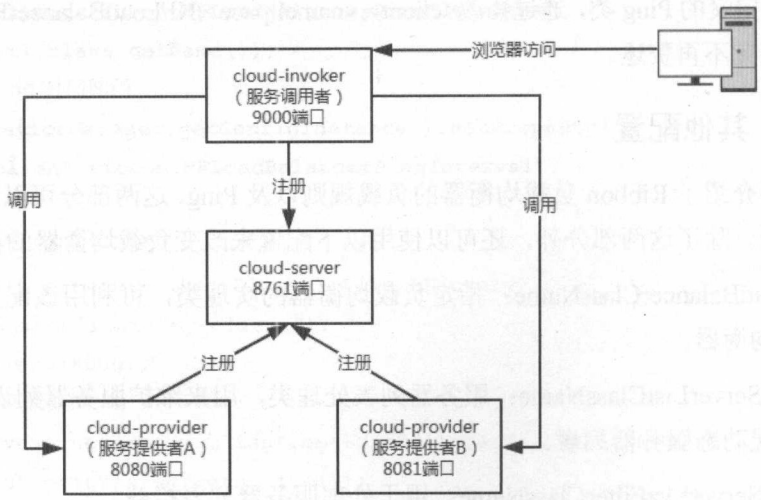


图 4-2 准备的项目结构图

Eureka 相关项目的建立可参见本书的 3.2 节。

4.4.2 使用代码配置 Ribbon

在 4.3 节中讲述了负载规则以及 Ping 机制，在 Spring Cloud 中，可将自定义的负载规则以及 Ping 类放到服务调用者中查看效果。新建自定义的 IRule 与 IPing，两个实现类请见代码清单 4-11。

代码清单 4-11：codes\04\4.4\cloud-invoker\src\main\java\org\crazyit\cloud\MyRule.java

codes\04\4.4\cloud-invoker\src\main\java\org\crazyit\cloud\MyPing.java

```
public class MyRule implements IRule {

    private ILoadBalancer lb;

    public Server choose(Object key) {
        List<Server> servers = lb.getAllServers();
        System.out.println("这是自定义服务器规则类，输出服务器信息：");
        for(Server s : servers) {
            System.out.println("        " + s.getHostPort());
        }
        return servers.get(0);
    }
    ……省略 setter 和 getter 方法
}
public class MyPing implements IPing {

    public boolean isAlive(Server server) {
        System.out.println("自定义 Ping 类，服务器信息：" + server.getHostPort());
        return true;
    }
}
```

根据两个自定义的 IRule 和 IPing 类可知，实际上跟 4.3 节中介绍的自定义实现类似，服务器选择规则只返回集合中的第一个实例，IPing 实现仅仅是控制输入服务器信息。接下来，新建配置类、返回规则与 Ping 的 Bean，请见代码清单 4-12。

代码清单 4-12：codes\04\4.4\cloud-invoker\src\main\java\org\crazyit\cloud\config\MyConfig.java
04\4.4\cloud-invoker\src\main\java\org\crazyit\cloud\config\CloudProviderConfig.java

```
public class MyConfig {
    @Bean
    public IRule getRule() {
        return new MyRule();
    }
    @Bean
    public IPing getPing() {
        return new MyPing();
    }
}

@RibbonClient(name="cloud-provider", configuration=MyConfig.class)
public class CloudProviderConfig {

}
```

在代码清单 4-12 中，CloudProviderConfig 配置类使用了@RibbonClient 注解，配置了 RibbonClient 的名称为 cloud-provider，对应的配置类为 MyConfig，也就是名称为 cloud-provider 的客户端将使用 MyRule 与 MyPing 两个类。在服务调用者的控制器中，加入对外服务，服务中调用 RestTemplate，如代码清单 4-13 所示。

代码清单 4-13：codes\04\4.4\cloud-invoker\src\main\java\org\crazyit\cloud\InvokerController.java

```
@RestController
@Configuration
public class InvokerController {

    @LoadBalanced
    @Bean
    public RestTemplate getRestTemplate() {
        return new RestTemplate();
    }

    @RequestMapping(value = "/router", method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON_VALUE)
    public String router() {
        RestTemplate restTpl = getRestTemplate();
```

```
        // 根据名称调用服务
        String json = restTpl.getForObject(
                "http://cloud-provider/person/1", String.class);
        return json;
    }
}
```

在以上的控制器中，为 RestTemplate 加入了@LoadBalanced 修饰，与第 3 章类似，在此不再赘述。关于 RestTemplate 的原理，将在本章后面的章节中讲述。进行以下操作，查看本例效果：

- 启动一个 Eureka 服务器（cloud-server）。
- 启动两次 Eureka 服务提供者（cloud-provider），分别输入 8080 与 8081 端口。
- 启动一个 Eureka 服务调用者（cloud-invoker）。
- 打开浏览器访问 http://localhost:9000/router，可以看到调用服务后返回的 JSON 字符串，不管刷新多少次，最终都只会访问其中一个端口。

4.4.3 使用配置文件设置 Ribbon

在前面使用 Ribbon 时，可以通过配置来定义各个属性。在使用 Spring Cloud 时，这些属性同样可以配置到 application.yml 中，以下的配置同样生效：

```
cloud-provider:
  ribbon:
    NFLoadBalancerRuleClassName: org.crazyit.cloud.MyRule
    NFLoadBalancerPingClassName: org.crazyit.cloud.MyPing
    listOfServers: http://localhost:8080/,http://localhost:8081/
```

为 cloud-provider 这个客户端配置了规则处理类、Ping 类以及服务器列表，以同样的方式运行本小节的例子，可看到同样的效果，在此不再赘述。

4.4.2 节使用了代码的方式来设置 Ribbon，而 4.4.3 节则使用配置文件的方式，两种方式的效果一样，但比较起来，明显是配置文件的方式更加简便。

注意：

在本案例的 cloud-invoker 模块中，默认使用了代码的方式来配置 Ribbon，配置文件中的配置已被注释。

4.4.4 Spring 使用 Ribbon 的 API

Spring Cloud 对 Ribbon 进行封装，例如像负载客户端、负载均衡器等，我们可以直接使用 Spring 的 LoadBalancerClient 来处理请求以及服务选择。代码清单 4-14 在服务器调用者的控制器中使用了 LoadBalancerClient。

代码清单 4-14：codes\04\4.4\cloud-invoker\src\main\java\org\crazyit\cloud\InvokerController.java

```
@Autowired
private LoadBalancerClient loadBalancer;

@RequestMapping(value = "/uselb", method = RequestMethod.GET,
        produces = MediaType.APPLICATION_JSON_VALUE)
public ServiceInstance uselb() {
    // 查找服务器实例
    ServiceInstance si = loadBalancer.choose("cloud-provider");
    return si;
}
```

除了使用 Spring 封装的负载客户端外，还可以直接使用 Ribbon 的 API，如代码清单 4-15 所示，直接获取 Spring Cloud 默认环境中各个 Ribbon 的实现类。

代码清单 4-15：codes\04\4.4\cloud-invoker\src\main\java\org\crazyit\cloud\InvokerController.java

```
@Autowired
private SpringClientFactory factory;

@RequestMapping(value = "/defaultValue", method = RequestMethod.GET,
        produces = MediaType.APPLICATION_JSON_VALUE)
public String defaultValue() {
    System.out.println("==== 输出默认配置: ");
    // 获取默认的配置
    ZoneAwareLoadBalancer alb = (ZoneAwareLoadBalancer) factory
            .getLoadBalancer("default");
    System.out.println("    IClientConfig: "
            + factory.getLoadBalancer("default").getClass()
                    .getName());
    System.out.println("    IRule: " + alb.getRule().getClass().getName());
    System.out.println("    IPing: " + alb.getPing().getClass().getName());
    System.out.println("    ServerList: "
            + alb.getServerListImpl().getClass().getName());
    System.out.println("    ServerListFilter: "
```

```
            + alb.getFilter().getClass().getName());
        System.out.println("    ILoadBalancer: " + alb.getClass().getName());
        System.out.println("    PingInterval: " + alb.getPingInterval());
        System.out.println("==== 输出 cloud-provider 配置: ");
        // 获取 cloud-provider 的配置
        ZoneAwareLoadBalancer alb2 = (ZoneAwareLoadBalancer) factory
                .getLoadBalancer("cloud-provider");
        System.out.println("    IClientConfig: "
                + factory.getLoadBalancer("cloud-provider").getClass().getName());
        System.out.println("    IRule: " + alb2.getRule().getClass().getName());
        System.out.println("    IPing: " + alb2.getPing().getClass().getName());
        System.out.println("    ServerList: "
                + alb2.getServerListImpl().getClass().getName());
        System.out.println("    ServerListFilter: "
                + alb2.getFilter().getClass().getName());
        System.out.println("    ILoadBalancer: " + alb2.getClass().getName());
        System.out.println("    PingInterval: " + alb2.getPingInterval());
        return "";
    }
```

代码中使用了 SpringClientFactory，通过该实例可获取各个默认的实现类以及配置，分别输出了默认配置以及 cloud-provider 配置。运行代码清单 4-15，在浏览器中访问地址 http://localhost:8080/defaultValue，可看到控制台中的输出如下：

```
==== 输出默认配置:
    IClientConfig: com.netflix.loadbalancer.ZoneAwareLoadBalancer
    IRule: com.netflix.loadbalancer.ZoneAvoidanceRule
    IPing: com.netflix.niws.loadbalancer.NIWSDiscoveryPing
    ServerList:
org.springframework.cloud.netflix.ribbon.eureka.DomainExtractingServerList
    ServerListFilter:
org.springframework.cloud.netflix.ribbon.ZonePreferenceServerListFilter
    ILoadBalancer: com.netflix.loadbalancer.ZoneAwareLoadBalancer
    PingInterval: 30
==== 输出 cloud-provider 配置:
    IClientConfig: com.netflix.loadbalancer.ZoneAwareLoadBalancer
    IRule: org.crazyit.cloud.MyRule
    IPing: org.crazyit.cloud.MyPing
    ServerList:
org.springframework.cloud.netflix.ribbon.eureka.DomainExtractingServerList
```

```
        ServerListFilter:
org.springframework.cloud.netflix.ribbon.ZonePreferenceServerListFilter
        ILoadBalancer: com.netflix.loadbalancer.ZoneAwareLoadBalancer
        PingInterval: 30
```

根据输出可知，cloud-provider 客户端使用的负载规则类以及 Ping 类，是我们自定义的实现类。

一般情况下，Spring 已经帮我们封装好了 Ribbon，我们只需直接调用 RestTemplate 等 API 来访问服务即可。在接下来的章节中，我们将讲述 RestTemplate 进行负载均衡的原理。

4.5 RestTemplate 负载均衡

4.5.1 @LoadBalanced 注解概述

RestTemplate 本是 spring-web 项目中的一个 REST 客户端，它遵循 REST 的设计原则，提供简单的 API 让我们去调用 HTTP 服务。RestTemplate 本身不具有负载均衡的功能，该类也与 Spring Cloud 没有关系，但为何加入@LoadBalanced 注解后，一个 RestTemplate 实例就具有负载均衡的功能了呢？实际上这要得益于 RestTemplate 的拦截器功能。

在 Spring Cloud 中，使用@LoadBalanced 修饰的 RestTemplate，在 Spring 容器启动时，会为这些被修饰过的 RestTemplate 添加拦截器，拦截器中使用了 LoadBalancerClient 来处理请求，LoadBalancerClient 本来就是 Spring 封装的负载均衡客户端，通过这样间接处理，使得 RestTemplate 拥有了负载均衡的功能。

本节将模仿拦截器机制，带领大家实现一个简单的 RestTemplate，以便让大家更了解@LoadBalanced 以及 RestTemplate 的原理。本节的案例只依赖了 spring-boot-starter-web 模块：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>1.5.4.RELEASE</version>
</dependency>
```

4.5.2 编写自定义注解以及拦截器

先模仿@LoadBalanced 注解，编写一个自定义注解，请见代码清单 4-16。

代码清单 4-16：codes\04\4.5\rest-template-test\src\main\java\org\crazyit\cloud\MyLoadBalanced.java

```
@Target({ ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface MyLoadBalanced {

}
```

注意，MyLoadBalanced 注解中使用了@Qualifier 限定注解，接下来编写自定义的拦截器，请见代码清单 4-17。

代码清单 4-17：codes\04\4.5\rest-template-test\src\main\java\org\crazyit\cloud\MyInterceptor.java

```
public class MyInterceptor implements ClientHttpRequestInterceptor {

    public ClientHttpResponse intercept(HttpRequest request, byte[] body,
            ClientHttpRequestExecution execution) throws IOException {
        System.out.println("============= 这是自定义拦截器实现");
        System.out.println("              原来的 URI: " + request.getURI());
        // 换成新的请求对象（更换 URI）
        MyHttpRequest newRequest = new MyHttpRequest(request);
        System.out.println("              拦截后新的 URI: " + request.getURI());
        return execution.execute(newRequest, body);
    }
}
```

在自定义拦截器 MyInterceptor 中，实现了 intercept 方法，该方法会将原来的 HttpRequest 对象转换为我们自定义的 MyHttpRequest，MyHttpRequest 是一个自定义的请求类，实现请见代码清单 4-18。

代码清单 4-18：codes\04\4.5\rest-template-test\src\main\java\org\crazyit\cloud\MyHttpRequest.java

```
public class MyHttpRequest implements HttpRequest {

    private HttpRequest sourceRequest;

    public MyHttpRequest(HttpRequest sourceRequest) {
        this.sourceRequest = sourceRequest;
    }

    public HttpHeaders getHeaders() {
        return sourceRequest.getHeaders();
```

```
    }

    public HttpMethod getMethod() {
        return sourceRequest.getMethod();
    }

    /**
     * 将URI转换
     */
    public URI getURI() {
        try {
            String oldUri = sourceRequest.getURI().toString();
            URI newUri = new URI("http://localhost:8080/hello");
            return newUri;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return sourceRequest.getURI();
    }
}
```

在MyHttpRequest类中，会将原来请求的URI进行改写，只要使用了这个对象，所有的请求都会被转发到http://localhost:8080/hello这个地址。**Spring Cloud在对RestTemplate进行拦截的时候也做了同样的事情，只不过并没有像我们这样固定了URI，而是对“源请求”进行了更加灵活的处理。**接下来使用自定义注解以及拦截器。

4.5.3 使用自定义拦截器以及注解

编写一个Spring的配置类，在初始化的Bean中为容器中的RestTemplate实例设置自定义拦截器，本例的Spring自动配置类请见代码清单4-19。

代码清单4-19：

codes\04\4.5\rest-template-test\src\main\java\org\crazyit\cloud\MyAutoConfiguration.java

```
@Configuration
public class MyAutoConfiguration {

    @Autowired(required=false)
    @MyLoadBalanced
    private List<RestTemplate> myTemplates = Collections.emptyList();
```

```
    @Bean
    public SmartInitializingSingleton myLoadBalancedRestTemplateInitializer() {
        System.out.println("====  这个 Bean 将在容器初始化时创建    =====");
        return new SmartInitializingSingleton() {

            public void afterSingletonsInstantiated() {
                for(RestTemplate tpl : myTemplates) {
                    // 创建一个自定义的拦截器实例
                    MyInterceptor mi = new MyInterceptor();
                    // 获取 RestTemplate 原来的拦截器
                    List list = new ArrayList(tpl.getInterceptors());
                    // 添加到拦截器集合
                    list.add(mi);
                    // 将新的拦截器集合设置到 RestTemplate 实例
                    tpl.setInterceptors(list);
                }
            }
        };
    }
}
```

在配置类中定义了 RestTemplate 实例的集合，并且使用了@MyLoadBalanced 以及@Autowired 注解进行修饰，@MyLoadBalanced 中含有@Qualifier 注解。简单来说，就是在 Spring 容器中使用了@MyLoadBalanced 修饰的 RestTemplate 实例，该实例将会被加入配置类的 RestTemplate 集合中。

在容器初始化时，Spring 会调用 myLoadBalancedRestTemplateInitializer 方法来创建 Bean，该 Bean 在初始化完成后，会遍历 RestTemplate 集合并为它们设置“自定义拦截器”，请见代码清单 4-19 中的粗体代码。下面在控制器中使用@MyLoadBalanced 来修饰调用者的 RestTemplate。

4.5.4 在控制器中使用 RestTemplate

控制器代码请见代码清单 4-20。

代码清单 4-20：codes\04\4.5\rest-template-test\src\main\java\org\crazyit\cloud\InvokerController.java

```
@RestController
@Configuration
```

```
public class InvokerController {

    @Bean
    @MyLoadBalanced
    public RestTemplate getMyRestTemplate() {
        return new RestTemplate();
    }

    /**
     * 浏览器访问的请求
     */
    @RequestMapping(value = "/router", method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON_VALUE)
    public String router() {
        RestTemplate restTpl = getMyRestTemplate();
        // 根据名称来调用服务，这个 URI 会被拦截器所置换
        String json = restTpl.getForObject("http://my-server/hello", String.class);
        return json;
    }

    /**
     * 最终的请求都会转到这个服务
     */
    @RequestMapping(value = "/hello", method = RequestMethod.GET)
    @ResponseBody
    public String hello() {
        return "Hello World";
    }
}
```

注意控制器的 hello 方法，前面实现的拦截器会将全部请求都转到这个服务中。控制器的 RestTemplate 使用了@MyLoadBalanced 注解进行修饰。熟悉前面使用 RestTemplate 的读者可发现，我们实现的注解与 Spring 提供的@LoadBalanced 注解使用方法一致。在控制器的 router 方法中，使用这个被拦截过的 RestTemplate 发送请求。

打开浏览器，访问 http://localhost:8080/router，可以看到实际上调用了 hello 服务。在访问该地址时，控制台输出如下：

```
============ 这是自定义拦截器实现
             原来的 URI：http://my-server/hello
             拦截后新的 URI：http://localhost:8080/hello
```

Spring Cloud 对 RestTemplate 的拦截实现更加复杂，并且在拦截器中使用 LoadBalancerClient 来实现请求的负载均衡功能。我们在实际环境中，并不需要实现自定义注解以及拦截器，用 Spring 提供的现成 API 即可，本节的目的是展示 RestTemplate 的原理。

4.6 本章小结

本章主要讲解了 Spring Cloud 中的负载均衡组件 Ribbon，对于 Ribbon 如何进行负载均衡进行了详细讲解。读者学习完本章后，可以了解 Ribbon 是如何进行负载均衡的，如果有需要，还可以实现自己的负载均衡规则，以满足实际环境中多变的需求。

作为 Spring Cloud 中的重要组件，Spring Cloud 对 Ribbon 进行了封装，在使用 Spring 提供的 API 时，我们甚至感觉不到 Ribbon 的存在。本章的 4.4 节，重点讲述了在 Spring Cloud 中如何配置和使用 Ribbon。

在很多章节中，我们都使用了 RestTemplate 发送请求，在 4.5 节，我们讲解了 RestTemplate 如何拥有负载均衡功能。读者学习完 4.5 节，可以更加清楚 RestTemplate 的工作机制，以及 Spring Cloud 对其进行的拦截处理。

CHAPTER

5

第5章 REST客户端Feign

本章要点

- REST 客户端
- Feign 框架的使用
- 在 Spring Cloud 中使用 Feign

在 Spring Cloud 集群中，各个角色的通信基于 REST 服务，因此在调用服务时，就不可避免地需要使用 REST 服务的请求客户端。前面的章节中使用了 Spring 自带的 RestTemplate，RestTemplate 使用 HttpClient 发送请求。本章中，我们将介绍另一个 REST 客户端：Feign。

Feign 框架已经被集成到 Spring Cloud 的 Netflix 项目中，使用该框架可以在 Spring Cloud 的集群中更加简单地调用 REST 服务。

5.1 REST 客户端

在学习 Feign 前，我们先了解一下 REST 客户端。本节将简单地讲述 Apache CXF 与 Restlet 这两款 Web Service 框架，并使用这两个框架来编写 REST 客户端，最后再编写一个 Feign 的 Hello World 例子。通过此过程，大家可以对 Feign 有一个初步的印象。如已经掌握这两个 REST 框架，可直接学习本章后面的内容。

本章中介绍的各个客户端，将会访问 8080 端口的/person/{personId}和/hello 这两个服务中的一个，服务端项目使用 spring-boot-starter-web 进行搭建，本节对应的服务端项目的目录为 codes\05\5.1\rest-server。

5.1.1 使用 CXF 调用 REST 服务

CXF 是目前一个较为流行的 Web Service 框架，是 Apache 的一个开源项目。使用 CXF 可以发布和调用使用各种协议的服务，包括 SOAP 协议、XML/HTTP 等。当前 CXF 已经对 REST 风格的 Web Service 提供支持，可以发布或调用 REST 风格的 Web Service。由于 CXF 可以与 Spring 进行整合使用并且配置简单，因此得到许多开发者的青睐，而笔者以往所在公司的大部分项目，均使用 CXF 来发布和调用 Web Service。本章所使用的 CXF 版本为 3.1.10，在 Maven 中加入以下依赖：

```
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-core</artifactId>
    <version>3.1.10</version>
</dependency>
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-rs-client</artifactId>
    <version>3.1.10</version>
</dependency>
```

编写代码请求/person/{personId}服务，请见代码清单 5-1。

代码清单 5-1：codes\05\5.1\rest-client\src\main\java\org\crazyit\cloud\CxfClient.java

```
public class CxfClient {

    public static void main(String[] args) throws Exception {
        // 创建 WebClient
        WebClient client = WebClient.create("http://localhost:8080/person/1");
        // 获取响应
        Response response = client.get();
        // 获取响应内容
        InputStream ent = (InputStream) response.getEntity();
        String content = IOUtils.readStringFromStream(ent);
        // 输出字符串
        System.out.println(content);
    }
}
```

客户端中使用了 WebClient 类发送请求，获取响应后读取输入流，获取服务返回的 JSON 字符串。运行代码清单 5-1，可看到返回的信息。

5.1.2 使用 Restlet 调用 REST 服务

Restlet 是一个轻量级的 REST 框架，使用它可以发布和调用 REST 风格的 Web Service。本小节中的例子所使用的版本为 2.3.10，Maven 依赖如下：

```
<dependency>
    <groupId>org.restlet.jee</groupId>
    <artifactId>org.restlet</artifactId>
    <version>2.3.10</version>
</dependency>
<dependency>
    <groupId>org.restlet.jee</groupId>
    <artifactId>org.restlet.ext.jackson</artifactId>
    <version>2.3.10</version>
</dependency>
```

客户端实现请见代码清单 5-2。

代码清单 5-2：codes\05\5.1\rest-client\src\main\java\org\crazyit\cloud\RestletClient.java

```
public class RestletClient {

    public static void main(String[] args) throws Exception {
        ClientResource client = new ClientResource(
                "http://localhost:8080/person/1");
        // 调用 get 方法，服务端发布的是 GET
        Representation response = client.get(MediaType.APPLICATION_JSON);
        // 创建 JacksonRepresentation 实例，将响应转换为 Map
        JacksonRepresentation jr = new JacksonRepresentation(response,
                HashMap.class);
        // 获取转换后的 Map 对象
        Map result = (HashMap) jr.getObject();
        // 输出结果
        System.out.println(result.get("id") + "-" + result.get("name") + "-"
                + result.get("age") + "-" + result.get("message"));
    }
}
```

代码清单 5-2 中使用的 Restlet 的 API 较为简单，在此不赘述。但需要注意的是，在 Maven 中使用 Restlet，要额外配置仓库地址，笔者成书时，在 Apache 官方仓库中并没有 Restlet 的包。在项目的 pom.xml 文件中增加以下配置：

```
<repositories>
    <repository>
        <id>maven-restlet</id>
        <name>Restlet repository</name>
        <url>http://maven.restlet.org</url>
    </repository>
</repositories>
```

5.1.3 Feign 框架介绍

Feign 是 GitHub 上的一个开源项目，目的是简化 Web Service 客户端的开发。在使用 Feign 时，可以使用注解来修饰接口，被注解修饰的接口具有访问 Web Service 的能力。这些注解中既包括 Feign 自带的注解，也支持使用第三方的注解。除此之外，Feign 还支持插件式的编码器和解码器，使用者可以通过该特性对请求和响应进行不同的封装与解析。

Spring Cloud 将 Feign 集成到 Netflix 项目中，当与 Eureka、Ribbon 集成时，Feign 就具有负载均衡的功能。Feign 本身在使用上的简便性，加上与 Spring Cloud 的高度整合，使用该框架在 Spring Cloud 中调用集群服务，将会大大降低开发的工作量。

5.1.4 第一个 Feign 程序

先使用 Feign 编写一个 Hello World 的客户端，访问服务端的/hello 服务，得到返回的字符串。当前 Spring Cloud 所依赖的 Feign 版本为 9.5.0，本章案例中的 Feign 也使用该版本。建立名称为 feign-client 的 Maven 项目，加入以下依赖：

```
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-core</artifactId>
    <version>9.5.0</version>
</dependency>
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-gson</artifactId>
    <version>9.5.0</version>
</dependency>
```

新建接口 HelloClient，请见代码清单 5-3。

代码清单 5-3：codes\05\5.1\feign-client\src\main\java\org\crazyit\cloud\HelloClient.java

```
public interface HelloClient {

    @RequestLine("GET /hello")
    String sayHello();
}
```

HelloClient 表示一个服务接口，在接口的 sayHello 方法中使用了@RequestLine 注解，表示使用 GET 方法向/hello 发送请求。接下来编写客户端的运行类，请见代码清单 5-4。

代码清单 5-4：codes\05\5.1\feign-client\src\main\java\org\crazyit\cloud\HelloMain.java

```
public class HelloMain {

    public static void main(String[] args) {
        // 调用 Hello 接口
```

```
        HelloClient hello = Feign.builder().target(HelloClient.class,
                "http://localhost:8080/");
        System.out.println(hello.sayHello());
    }
}
```

在运行类中，使用 Feign 创建 HelloClient 接口的实例，最后调用接口定义的方法。运行代码清单 5-4，可以看到返回的“Hello World”字符串，可见接口已经被调用。熟悉 AOP 的朋友大概已经猜到，Feign 实际上会帮我们动态生成代理类。Feign 使用的是 JDK 的动态代理，生成的代理类会将请求的信息封装，交给 feign.Client 接口发送请求，而该接口的默认实现类最终会使用 java.net.HttpURLConnection 来发送 HTTP 请求。

5.1.5 请求参数与返回对象

本案例中有两个服务，另外一个地址为/person/{personId}，需要传入参数并且返回 JSON 字符串。编写第二个 Feign 客户端，调用该服务。新建 PersonClient 服务类，定义调用接口并添加注解，请见代码清单 5-5。

代码清单 5-5：codes\05\5.1\feign-client\src\main\java\org\crazyit\cloud\PersonClient.java

```
public interface PersonClient {

    @RequestLine("GET /person/{personId}")
    Person findById(@Param("personId") Integer personId);

    @Data // 为所有属性加上 setter 和 getter 等方法
    class Person {
        Integer id;
        String name;
        Integer age;
        String message;
    }
}
```

定义的接口名称为 findById，参数为 personId。需要注意的是，由于会返回 Person 实例，我们在接口中定义了一个 Person 的类；为了减少代码量，使用了 Lombok 项目，使用了该项目的@Data 注解。要使用 Lombok，需要添加以下 Maven 依赖：

```
<dependency>
    <groupId>org.projectlombok</groupId>
```

```
    <artifactId>lombok</artifactId>
    <version>1.16.18</version>
</dependency>
```

准备好提供服务的客户端类后，再编写运行类。运行类基本上与前面的 Hello World 类似，请见代码清单 5-6。

代码清单 5-6：codes\05\5.1\feign-client\src\main\java\org\crazyit\cloud\PersonMain.java

```
public class PersonMain {

    public static void main(String[] args) {
        PersonClient personService = Feign.builder()
                .decoder(new GsonDecoder())
                .target(PersonClient.class, "http://localhost:8080/");
        Person person = personService.findById(2);
        System.out.println(person.id);
        System.out.println(person.name);
        System.out.println(person.age);
        System.out.println(person.message);
    }
}
```

在调用 Person 服务的运行类中添加了解码器的配置，GsonDecoder 会将返回的 JSON 字符串转换为接口方法返回的对象，关于解码器的内容，将在后面章节中讲述。运行代码清单 5-6，可以看到最终的输出。

本节使用了 CXF、Restlet、Feign 来编写 REST 客户端，在编写客户端的过程中，可以看到 Feign 的代码更加“面向对象”，至于是否更加简洁，则见仁见智。下面的章节，将深入了解 Feign 的各项功能。

5.2 使用 Feign

本节所有的案例都是单独使用 Feign，Feign 在 Spring Cloud 中的使用将在 5.3 节讲述，请读者注意该细节。服务端的项目，以 5.1 节的 rest-server 项目为基础，该项目是一个 Spring Boot Web 项目。

5.2.1 编码器

向服务发送请求的过程中，有些情况需要对请求的内容进行处理。例如服务端发布的服务接收的是 JSON 格式的参数，而客户端使用的是对象，这种情况就可以使用编码器，将对象转换为 JSON 字符串。

为服务端编写一个 REST 服务，处理 POST 请求，请见代码清单 5-7。

代码清单 5-7：codes\05\5.1\rest-server\src\main\java\org\crazyit\cloud\MyController.java

```
@RequestMapping(value = "/person/create", method = RequestMethod.POST,
        consumes = MediaType.APPLICATION_JSON_VALUE)
public String createPerson(@RequestBody Person person) {
    System.out.println(person.getName() + "-" + person.getAge());
    return "Success, Person Id: " + person.getId();
}
```

在控制器中发布了一个/person/create 服务，需要传入 JSON 格式的请求参数。在客户端中，要调用该服务，先编写接口，再使用注解进行修饰，请见代码清单 5-8。

代码清单 5-8：codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\PersonClient.java

```
public interface PersonClient {

    @RequestLine("POST /person/create")
    @Headers("Content-Type: application/json")
    String createPerson(Person person);

    @Data
    class Person {
        Integer id;
        String name;
        Integer age;
        String message;
    }
}
```

注意，在客户端的服务接口中，使用了@Headers 注解，声明请求的内容类型为 JSON，接下来再编写运行类，如代码清单 5-9 所示。

代码清单 5-9：codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\EncoderTest.java

```
public class EncoderTest {
```

```
    public static void main(String[] args) {
        // 获取服务接口
        PersonClient personClient = Feign.builder()
                .encoder(new GsonEncoder())
                .target(PersonClient.class, "http://localhost:8080/");
        // 创建参数的实例
        Person person = new Person();
        person.id = 1;
        person.name = "Angus";
        person.age = 30;
        String response = personClient.createPerson(person);
        System.out.println(response);
    }
}
```

在运行类中，在创建服务接口实例时，使用了 encoder 方法来指定编码器，本案例使用了 Feign 提供的 GsonEncoder 类。该类会在发送请求的过程中，将请求的对象转换为 JSON 字符串。Feign 支持插件式的编码器，如果 Feign 提供的编码器无法满足要求，还可以使用自定义的编码器，这部分内容在后面章节讲述。启动服务，运行代码清单 5-9，可看到服务已经调用成功，运行后输出如下：

```
Success, Person Id: 1
```

5.2.2 解码器

编码器是对请求的内容进行处理，解码器则会对服务响应的内容进行处理，例如将解析响应的 JSON 或者 XML 字符串，转换为我们所需要的对象，在代码中通过以下代码片断设置解码器：

```
PersonClient personService = Feign.builder()
    .decoder(new GsonDecoder())
    .target(PersonClient.class, "http://localhost:8080/");
```

在前面的章节中，我们已经使用过 GsonDecoder 解码器，在此不再赘述。

5.2.3 XML 的编码与解码

除了支持 JSON 的处理外，Feign 还为 XML 的处理提供了编码器与解码器，可以使用 JAXBEncoder 与 JAXBDecoder 进行编码与解码。为服务端添加发布 XML 的接口，请见代码清单 5-10。

代码清单 5-10：codes\05\5.1\rest-server\src\main\java\org\crazyit\cloud\MyController.java

```
@RequestMapping(value = "/person/createXML", method = RequestMethod.POST,
        consumes = MediaType.APPLICATION_XML_VALUE,
        produces = MediaType.APPLICATION_XML_VALUE)
public String createXMLPerson(@RequestBody Person person) {
    System.out.println(person.getName() + "-" + person.getId());
    return "<result><message>success</message></result>";
}
```

在服务端发布的服务方法中，声明了传入的参数为 XML。需要注意的是，服务端项目 rest-server 使用 spring-boot-starter-web 进行构建，默认情况下不支持 XML 接口，调用接口时会得到以下异常信息：

```
{"timestamp":1502705981406,"status":415,"error":"Unsupported Media Type",
"exception":"org.springframework.web.HttpMediaTypeNotSupportedException","message"
:"Content type 'application/xml;charset=UTF-8' not
supported","path":"/person/createXML"}
```

为服务端的 pom.xml 加入以下依赖即可解决该问题：

```
<dependency>
    <groupId>com.fasterxml.jackson.jaxrs</groupId>
    <artifactId>jackson-jaxrs-xml-provider</artifactId>
    <version>2.9.0</version>
</dependency>
```

编写客户端时，先定义好服务接口以及对象，接口请见代码清单 5-11。

代码清单 5-11：codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\PersonClient.java

```
public interface PersonClient {

    @RequestLine("POST /person/createXML")
    @Headers("Content-Type: application/xml")
    Result createPersonXML(Person person);

    @Data
    @XmlRootElement
    class Person {
        @XmlElement
        Integer id;
        @XmlElement
        String name;
```

```
        @XmlElement
        Integer age;
        @XmlElement
        String message;
    }

    @Data
    @XmlRootElement
    class Result {
        @XmlElement
        String message;
    }
}
```

在接口中，定义了 Content-Type 为 XML，使用了 JAXB 的相关注解来修饰 Person 与 Result。接下来，只需调用 createPersonXML 方法即可请求服务，请见代码清单 5-12。

代码清单 5-12：codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\XMLTest.java

```
public class XMLTest {

    public static void main(String[] args) {
        JAXBContextFactory jaxbFactory = new JAXBContextFactory.Builder().build();
        // 获取服务接口
        PersonClient personClient = Feign.builder()
                .encoder(new JAXBEncoder(jaxbFactory))
                .decoder(new JAXBDecoder(jaxbFactory))
                .target(PersonClient.class, "http://localhost:8080/");
        // 构建参数
        Person person = new Person();
        person.id = 1;
        person.name = "Angus";
        person.age = 30;
        // 调用接口并返回结果
        Result result = personClient.createPersonXML(person);
        System.out.println(result.message);
    }
}
```

本小节的请求有一点特殊，请求服务时传入的参数为XML的、返回的结果也是XML的，目的是使编码与解码一起使用。开启服务，运行代码清单5-12，可以看到服务端与客户端的输出。

5.2.4 自定义编码器与解码器

根据前面两小节的介绍可知，Feign的插件式编码器与解码器可以对请求以及结果进行处理。对于一些特殊的要求，可以使用自定义的编码器与解码器。实现自定义编码器，需要实现Encoder接口的encode方法，而对于解码器，则要实现Decoder接口的decode方法，例如以下的代码片断：

```
public class MyEncoder implements Encoder {

    public void encode(Object object, Type bodyType, RequestTemplate template)
            throws EncodeException {
        // 实现自己的Encode逻辑
    }
}
```

在使用时，调用Feign的API来设置编码器或者解码器即可，实现较为简单，在此不再赘述。

5.2.5 自定义Feign客户端

Feign使用一个Client接口来发送请求，默认情况下，使用HttpURLConnection连接HTTP服务。与前面的编码器类似，客户端也采用插件式设计，也就是说，我们可以实现自己的客户端。本小节将使用HttpClient来实现一个简单的Feign客户端。为pom.xml加入HttpClient的依赖：

```
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.5.2</version>
</dependency>
```

新建feign.Client接口的实现类，具体实现请见代码清单5-13。

代码清单5-13：codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\MyFeignClient.java

```
public class MyFeignClient implements Client {
```

```
    public Response execute(Request request, Options options)
            throws IOException {
        System.out.println("====  这是自定义的 Feign 客户端");
        try {
            // 创建一个默认的客户端
            CloseableHttpClient httpclient = HttpClients.createDefault();
            // 获取调用的 HTTP 方法
            final String method = request.method();
            // 创建一个 HttpClient 的 HttpRequest
            HttpRequestBase httpRequest = new HttpRequestBase() {
                public String getMethod() {
                    return method;
                }
            };
            // 设置请求地址
            httpRequest.setURI(new URI(request.url()));
            // 执行请求，获取响应
            HttpResponse httpResponse = httpclient.execute(httpRequest);
            // 获取响应的主体内容
            byte[] body = EntityUtils.toByteArray(httpResponse.getEntity());
            // 将 HttpClient 的响应对象转换为 Feign 的 Response
            Response response = Response.builder()
                    .body(body)
                    .headers(new HashMap<String, Collection<String>>())
                    .status(httpResponse.getStatusLine().getStatusCode())
                    .build();
            return response;
        } catch (Exception e) {
            throw new IOException(e);
        }
    }
}
```

简单讲一下自定义 Feign 客户端的实现过程。在实现 execute 方法时，将 Feign 的 Request 实例转换为 HttpClient 的 HttpRequestBase。再使用 CloseableHttpClient 来执行请求，得到响应的 HttpResponse 实例后，再转换为 Feign 的 Response 实例返回。我们实现的客户端，包括 Feign 自带的客户端以及其他扩展的客户端，实际上就是一个对象转换的过程。在运行

类中直接使用我们自定义的客户端，请见代码清单 5-14。

代码清单 5-14：codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\MyClientTest.java

```
public class MyClientTest {

    public static void main(String[] args) {
        // 获取服务接口
        PersonClient personClient = Feign.builder()
                .encoder(new GsonEncoder())
                .client(new MyFeignClient())
                .target(PersonClient.class, "http://localhost:8080/");
        // 请求 Hello World 接口
        String result = personClient.sayHello();
        System.out.println("    接口响应内容：" + result);
    }
}
```

运行代码清单 5-14，输出如下：

```
====  这是自定义的 Feign 客户端
    接口响应内容：Hello World
```

注意：

在本例的实现中，笔者简化了实现，自定义的客户端中并没有转换请求头等信息，因此使用本例的客户端，无法请求其他格式的服务。

虽然 Feign 也有 HttpClient 的实现，但本例的目的主要是向大家展示 Feign 客户端的原理。举一反三，如果我们实现一个客户端，在实现中调用 Ribbon 的 API 来实现负载均衡的功能，是完全可以实现的。幸运的是，Feign 已经帮我们实现了 RibbonClient，可以直接使用，更进一步，Spring Cloud 也实现了自己的 Client，我们将在后面章节中讲述。

5.2.6 使用第三方注解

根据前面章节的介绍可知，通过注解修改的接口方法，可以让接口方法获得访问服务的能力。除了 Feign 自带的方法外，还可以使用第三方的注解。如果想使用 JAXRS 规范的注解，可以使用 Feign 的 feign-jaxrs 模块，在 pom.xml 中加入以下依赖即可：

```
<!-- Feign 对 JAXRS 的支持 -->
```

```
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-jaxrs</artifactId>
    <version>9.5.0</version>
</dependency>
<!-- JAXRS -->
<dependency>
    <groupId>javax.ws.rs</groupId>
    <artifactId>jsr311-api</artifactId>
    <version>1.1.1</version>
</dependency>
```

在使用注解修饰接口时，可以直接使用@GET、@Path 等注解，例如想要使用 GET 方法调用/hello 服务，可以定义以下接口：

```
@GET @Path("/hello")
String rsHello();
```

以上修饰接口的，实际上等价于@RequestLine("GET /hello")。为了让 Feign 知道这些注解的作用，需要在创建服务客户端时调用 contract 方法来设置 JAXRS 注解的解析类，请见以下代码：

```
RSClient rsClient = Feign.builder()
    .contract(new JAXRSContract())
    .target(RSClient.class, "http://localhost:8080/");
```

设置了 JAXRSContract 类后，Feign 就知道如何处理 JAXRS 的相关注解了。下一小节，我们将讲解 Feign 是如何处理第三方注解的。

5.2.7 Feign 解析第三方注解

根据前一小节的介绍可知，设置了 JAXRSContract 后，Feign 就知道如何处理接口中的 JAXRS 注解了。JAXRSContract 继承了 BaseContract 类，BaseContract 类实现了 Contract 接口，简单来说，一个 Contract 就相当于一个**翻译器**，Feign 本身并不知道这些第三方注解的含义，而通过实现一个翻译器（Contract）来告诉 Feign，这些注解是做什么的。

为了让读者能够了解其中的原理，本小节将使用一个自定义注解，并且翻译给 Feign，让其去使用。代码清单 5-15 所示为自定义注解以及客户端接口的代码。

代码清单 5-15：codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\contract\MyUrl.java
codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\contract\HelloClient.java

```
@Target(METHOD)
@Retention(RUNTIME)
public @interface MyUrl {

    // 定义 url 与 method 属性
    String url();
    String method();
}

public interface HelloClient {

    @MyUrl(method = "GET", url = "/hello")
    String myHello();
}
```

接下来，就要将 MyUrl 注解的作用告诉 Feign，新建 Contract 继承 BaseContract 类，实现请见代码清单 5-16。

代码清单 5-16：codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\contract\MyContract.java

```
public class MyContract extends Contract.BaseContract {

    @Override
    protected void processAnnotationOnClass(MethodMetadata data, Class<?> clz) {

    }

    /**
     * 用于处理方法级的注解
     */
    protected void processAnnotationOnMethod(MethodMetadata data,
            Annotation annotation, Method method) {
        // 是 MyUrl 注解才进行处理
        if(MyUrl.class.isInstance(annotation)) {
            // 获取注解的实例
            MyUrl myUrlAnn = method.getAnnotation(MyUrl.class);
            // 获取配置的 HTTP 方法
            String httpMethod = myUrlAnn.method();
```

```
            // 获取服务的 url
            String url = myUrlAnn.url();
            // 将值设置到模板中
            data.template().method(httpMethod);
            data.template().append(url);
        }
    }

    @Override
    protected boolean processAnnotationsOnParameter(MethodMetadata data,
            Annotation[] annotations, int paramIndex) {
        return false;
    }
}
```

在 MyContract 类中，需要实现三个方法，分别是处理类注解、处理方法注解、处理参数注解的方法，由于我们只定义了一个方法注解 @MyUrl，因此实现 processAnnotationOnMethod 即可。

在 processAnnotationOnMethod 方法中，通过 Method 的 getAnnotation 获取 MyUrl 的实例，将 MyUrl 的 url、method 属性分别设置到 Feign 的模板中。在创建客户端时，再调用 contract 方法即可，请见代码清单 5-17。

代码清单 5-17：codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\contract\ContractTest.java

```
public class ContractTest {

    public static void main(String[] args) {
        // 获取服务接口
        HelloClient helloClient = Feign.builder()
                .contract(new MyContract())
                .target(HelloClient.class, "http://localhost:8080/");
        // 请求 Hello World 接口
        String result = helloClient.myHello();
        System.out.println("    接口响应内容：" + result);
    }
}
```

运行代码清单 5-18，可看到控制台输出如下：

```
    接口响应内容：Hello World
```

由本例可知，Contract 实际上承担的是翻译的作用，将第三方（或者自定义）注解的作用告诉 Feign。在 Spring Cloud 中，也实现了 Spring 的 Contract，可以在接口中使用@RequestMapping 注解。读者在学习 Spring Cloud 整合 Feign 时，见到使用@RequestMapping 修饰的接口，就可以明白其中的原理。

5.2.8 请求拦截器

Feign 支持请求拦截器，在发送请求前，可以对发送的模板进行操作，例如设置请求头的属性等。自定义请求拦截器，实现 RequestInterceptor 接口，在创建客户端时，调用相应的方法设置一个或者多个拦截器，请见以下代码片断：

```
// 自定义拦截器
public class MyInterceptor implements RequestInterceptor {

    public void apply(RequestTemplate template) {
        template.header("Content-Type", "application/json");
    }
}
// 设置请求拦截器
public class InterceptorTest {

    public static void main(String[] args) {
        // 获取服务接口
        PersonClient personClient = Feign.builder()
                .requestInterceptor(new MyInterceptor())
                .target(PersonClient.class, "http://localhost:8080/");
    }
}
```

在使用时，根据实际情况进行设置即可，在此不再赘述。

5.2.9 接口日志

默认情况下，不会记录接口的日志，如果需要很清楚地了解接口的调用情况，可以使用 logLevel 方法进行配置，请见以下代码：

```
// 获取服务接口
PersonClient personClient = Feign.builder()
    .logLevel(Logger.Level.FULL)
```

```
        .logger(new Logger.JavaLogger().appendToFile("logs/http.log"))
        .target(PersonClient.class, "http://localhost:8080/");
    personClient.sayHello();
```

调用了 logLevel 设置接口的日志级别，调用了 logger 方法设置日志记录方式，本例是输出到文件中，运行以上代码，再打开日志文件，可以看到接口的日志如下：

```
[PersonClient#sayHello] ---> GET http://localhost:8080/hello HTTP/1.1
[PersonClient#sayHello] ---> END HTTP (0-byte body)
[PersonClient#sayHello] <--- HTTP/1.1 200 (110ms)
[PersonClient#sayHello] content-length: 11
[PersonClient#sayHello] content-type: text/plain;charset=UTF-8
[PersonClient#sayHello] date: Wed, 16 Aug 2017 14:58:58 GMT
[PersonClient#sayHello]
[PersonClient#sayHello] Hello World
[PersonClient#sayHello] <--- END HTTP (11-byte body)
```

以上日志，记录的就是一次请求的过程。设置接口的日志级别，有以下可选值。

- NONE：默认值，不进行日志记录。
- BASIC：记录请求方法、URL、响应状态代码和执行时间。
- HEADERS：除了 BASIC 记录的信息外，还包括请求头与响应头。
- FULL：记录全部日志，包括请求头、请求体、请求与响应的元数据。

记录接口日志的调用过程可以很方便地查找问题，不管在开发环境还是生产环境，都有较大的意义。

5.3 在 Spring Cloud 中使用 Feign

前一节讲解了 Feign 的使用，在了解了如何单独使用 Feign 后，再学习在 Spring Cloud 中使用 Feign，将会有非常大的帮助。虽然 Spring Cloud 对 Feign 进行了封装，但万变不离其宗，只要了解其内在原理，使用起来就可以得心应手。

在开始本节的讲解前，先准备 Spring Cloud 的测试项目。测试案例主要有以下三个项目。

- spring-feign-server：Eureka 服务器端项目，端口为 8761，代码目录为 codes\05\5.3\spring-feign-server。

- spring-feign-provider：服务提供者，代码目录为 codes\05\5.3\spring-feign-provider，该项目可以在控制台中根据输入的端口号启动多个实例，启动 8080 与 8081 这两个端口，该项目提供以下两个 REST 服务。
  - 第一个地址为/person/{personId}的服务，请求后返回 Person 实例，Person 的 message 属性为 HTTP 请求的 URL。
  - 第二个地址为/hello 的服务，返回“Hello World”字符串。
- spring-feign-invoker：服务调用者项目，对外端口为 9000，代码目录为 codes\05\5.3\spring-feign-invoker，本节的例子主要在该项目下使用 Feign。

5.3.1 Spring Cloud 整合 Feign

为服务调用者（spring-feign-invoker）的 pom.xml 文件加入以下依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

在服务调用者的启动类中，打开 Feign 开关，请见代码清单 5-18。

代码清单 5-18：

codes\05\5.3\spring-feign-invoker\src\main\java\org\crazyit\cloud\InvokerApplication.java

```
@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients
public class InvokerApplication {

    public static void main(String[] args) {
        SpringApplication.run(InvokerApplication.class, args);
    }
}
```

接下来，编写客户端接口，与直接使用 Feign 类似，代码清单 5-19 所示为服务端接口。

代码清单 5-19：codes\05\5.3\spring-feign-invoker\src\main\java\org\crazyit\cloud\PersonClient.java

```
@FeignClient("spring-feign-provider") //声明调用的服务名称
public interface PersonClient {
```

```
    @RequestMapping(method = RequestMethod.GET, value = "/hello")
    String hello();
}
```

与单独使用 Feign 不同的是，接口使用了@FeignClient 注解来修饰，并且声明了需要调用的服务名称，本例的服务提供者名称为 spring-feign-provider。另外，接口方法使用了@RequestMapping 来修饰，根据 5.2.7 节的介绍可知，通过编写“翻译器（Contract）”，可以让 Feign 知道第三方注解的含义，Spring Cloud 也提供翻译器，会将@RequestMapping 注解的含义告知 Feign，因此我们的服务接口就可以直接使用该注解。

除了方法的@RequestMapping 注解外，默认还支持@RequestParam、@RequestHeader、@PathVariable 这 3 个参数注解，也就是说，在定义方法时，可以使用以下方式定义参数：

```
@RequestMapping(method = RequestMethod.GET, value = "/hello/{name}")
String hello(@PathVariable("name") String name);
```

需要注意的是，使用了 Spring Cloud 的“翻译器”后，将不能再使用 Feign 的默认注解。接下来，在控制器中调用接口方法，请见代码清单 5-20。

代码清单 5-20：codes\05\5.3\spring-feign-invoker\src\main\java\org\crazyit\cloud\InvokerController.java

```
@RestController
@Configuration
public class InvokerController {

    @Autowired
    private PersonClient personClient;

    @RequestMapping(value = "/invokeHello", method = RequestMethod.GET)
    public String invokeHello() {
        return personClient.hello();
    }
}
```

在控制器中，为其注入了 PersonClient 的 Bean，不难看出，客户端实例的创建及维护，Spring 容器都帮我们实现了。查看本例的效果，请按以下步骤操作：

- 启动 Eureka 服务器（spring-feign-server）。
- 启动两个服务提供者(spring-feign-provider)，在控制台中分别输入 8080 与 8081 端口。
- 启动一个服务调用者（spring-feign-invoker），端口为 9000。

- 在浏览器中输入 http://localhost:9000/invokeHello，可以看到服务提供者的/hello 服务被调用。

5.3.2 Feign 负载均衡

在 5.2 节，我们尝试过编写自定义的 Feign 客户端，在 Spring Cloud 中，同样提供了自定义的 Feign 客户端。大家可能已经猜到，如果结合 Ribbon 使用，Spring Cloud 所提供的客户端会拥有负载均衡的功能。

Spring Cloud 实现的 Feign 客户端，类名为 LoadBalancerFeignClient，在该类中，维护着与 SpringClientFactory 相关的实例。通过 SpringClientFactory 可以获取负载均衡器，负载均衡器会根据一定的规则来选取处理请求的服务器，最终实现负载均衡的功能。接下来，调用服务提供者的/person/{personId}服务来测试负载均衡，为客户端接口添加内容，请见代码清单 5-21。

代码清单 5-21：codes\05\5.3\spring-feign-invoker\src\main\java\org\crazyit\cloud\PersonClient.java

```
@RequestMapping(method = RequestMethod.GET, value = "/person/{personId}")
Person getPerson(@PathVariable("personId") Integer personId);
```

为服务调用者的控制器添加方法，请见如下代码清单。

```
@RequestMapping(value = "/router", method = RequestMethod.GET,
        produces = MediaType.APPLICATION_JSON_VALUE)
public String router() {
    // 调用服务提供者的接口
    Person p = personClient.getPerson(2);
    return p.getMessage();
}
```

运行服务调用者，在浏览器中输入 http://localhost:9000/router，刷新，可以看到 8080 与 8081 端口被循环调用。

5.3.3 默认配置

Spring Cloud 为 Feign 的使用提供了各种默认属性，例如前面讲到的注解翻译器（Contract）、Feign 客户端。默认情况下，Spring 将会为 Feign 的属性提供以下的 Bean。

- 解码器（Decoder）：Bean 名称为 feignDecoder，ResponseEntityDecoder 类。
- 编码器（Encoder）：Bean 名称为 feignEncoder，SpringEncoder 类。

- 日志（Logger）：Bean 名称为 feignLogger，Slf4jLogger 类。
- 注解翻译器（Contract）：Bean 名称为 feignContract，SpringMvcContract 类。
- Feign 实例的创建者（Feign.Builder）：Bean 名称为 feignBuilder，HystrixFeign.Builder 类。Hystrix 框架将在后面章节中讲述。
- Feign 客户端（Client）：Bean 名称为 feignClient，LoadBalancerFeignClient 类。

一般情况下，Spring 提供的这些 Bean 已经足够我们使用，如果有些更特殊的需求，可以实现自己的 Bean，请见下一小节。

5.3.4 自定义配置

如果需要使用自己提供的 Feign 实现，可以在 Spring 的配置类中返回对应的 Bean，下面自定义一个简单的注解翻译器，代码清单 5-22 是一个配置类。

代码清单 5-22：codes\05\5.3\spring-feign-invoker\src\main\java\org\crazyit\cloud\contract\MyConfig.java

```
@Configuration
public class MyConfig {

    /**
     * 返回一个自定义的注解翻译器
     */
    @Bean
    public Contract feignContract() {
        return new MyContract();
    }
}
```

配置类中返回了一个 MyContract 实例，MyContract 是我们自定义的“翻译器”，实现请见代码清单 5-23。

代码清单 5-23：

codes\05\5.3\spring-feign-invoker\src\main\java\org\crazyit\cloud\contract\MyContract.java

```
/**
 * 自定义 Contract
 * @author 杨恩雄
 */
public class MyContract extends SpringMvcContract {
```

```
    /**
     * 用于处理方法级的注解
     */
    protected void processAnnotationOnMethod(MethodMetadata data,
            Annotation annotation, Method method) {
        // 调用父类的方法，让其支持 @RequestMapping 注解
        super.processAnnotationOnMethod(data, annotation, method);
        // 是 MyUrl 注解才进行处理
        if(MyUrl.class.isInstance(annotation)) {
            // 获取注解的实例
            MyUrl myUrlAnn = method.getAnnotation(MyUrl.class);
            // 获取配置的 HTTP 方法
            String httpMethod = myUrlAnn.method();
            // 获取服务的 url
            String url = myUrlAnn.url();
            // 将值设置到模板中
            data.template().method(httpMethod);
            data.template().append(url);
        }
    }
}
```

在前面的章节中，我们也实现过自定义的 Contract，与前面实现的 Contract 不同的是，本例的 MyContract 继承了 SpringMvcContract，在重写 processAnnotationOnMethod 方法时，调用了父类的 processAnnotationOnMethod。简单点说，我们实现的这个 Contract，除了支持 Spring 的注解外，还支持我们自定义的@MyUrl 注解。@MyUrl 注解与前面章节中介绍的一致，请见代码清单 5-24。

代码清单 5-24：codes\05\5.3\spring-feign-invoker\src\main\java\org\crazyit\cloud\contract\MyUrl.java

```
@Target(METHOD)
@Retention(RUNTIME)
public @interface MyUrl {

    // 定义 url 与 method 属性
    String url();
    String method();
}
```

接下来，编写客户端接口，可以使用 Spring 的@RequestMapping，或者是我们自定义的@MyUrl 注解，代码清单 5-25 所示为客户端接口。

代码清单 5-25：

codes\05\5.3\spring-feign-invoker\src\main\java\org\crazyit\cloud\contract\HelloClient.java

```
@FeignClient(name = "spring-feign-provider")
public interface HelloClient {

    @MyUrl(method = "GET", url = "/hello")
    String myHello();

    @RequestMapping(method = RequestMethod.GET, value = "/hello")
    String springHello();
}
```

在客户端接口中，分别使用了两个注解来调用同一个服务，接下来，在控制器中使用 HelloClient。

代码清单 5-26：codes\05\5.3\spring-feign-invoker\src\main\java\org\crazyit\cloud\InvokerController.java

```
@Autowired
private HelloClient helloClient;

@RequestMapping(value = "/testContract", method = RequestMethod.GET,
        produces = MediaType.APPLICATION_JSON_VALUE)
@ResponseBody
public String testContract() {
    String springResult = helloClient.springHello();
    System.out.println("使用 @RequestMapping 注解的接口返回结果：" + springResult);
    String myResult = helloClient.myHello();
    System.out.println("使用 @MyUrl 注解的接口返回结果：" + myResult);
    return "";
}
```

向控制器注入客户端接口，testContract 方法中分别调用两个 hello 方法。启动集群，访问服务调用者的地址 http://localhost:9000/testContract，可以看到控制台的输出如下：

```
使用 @RequestMapping 注解的接口返回结果：Hello World
使用 @MyUrl 注解的接口返回结果：Hello World
```

除了自定义的注解翻译器外，还可以自定义其他的 Bean，实现过程基本一致，在此不再赘述。

5.3.5 可选配置

在5.3.3节中介绍了若干个配置，Spring为这些配置提供了默认的Bean。除了这些配置外，还有如下的配置，Spring并没有提供默认的Bean。

- Logger.Level：接口日志的记录级别，相当于调用了Feign.Builder的logLevel方法，请见5.2.9节。
- Retryer：重试处理器，相当于调用了Feign.Builder的retryer方法。
- ErrorDecoder：异常解码器，相当于调用了Feign.Builder的errorDecoder方法。
- Request.Options：设置请求的配置项，相当于调用了Feign.Builder的options方法。
- Collection<RequestInterceptor>：设置请求拦截器，相当于调用了Feign.Builder的requestInterceptors方法。
- SetterFactory：该配置与Hystrix框架相关，将在后面章节详细讲述。

以上的配置，如果没有提供对应的Bean，则不会被设置。在此需要注意的是请求拦截器，由于可以设置多个请求拦截器，在创建Bean时也可以创建多个，返回类型需要为RequestInterceptor或者实现类。要设置多个请求拦截器，请见以下代码片断：

```
@Bean
public RequestInterceptor getRequestInterceptorsA() {
    return new RequestInterceptor() {
        public void apply(RequestTemplate template) {
            System.out.println("这是第一个请求拦截器");
        }
    };
}

@Bean
public RequestInterceptor getRequestInterceptorsB() {
    return new RequestInterceptor() {
        public void apply(RequestTemplate template) {
            System.out.println("这是第二个请求拦截器");
        }
    };
}
```

实现其他可选配置的Bean较为简单，不再赘述。

5.3.6 压缩配置

Feign 支持对请求和响应进行压缩处理，默认使用 GZIP 进行压缩，压缩操作在 Feign 的请求拦截器中实现。可以在配置文件中加入以下配置。

- feign.compression.request.enabled：设置为 true 开启请求压缩。
- feign.compression.response.enabled：设置为 true 开启响应压缩。
- feign.compression.request.mime-types：数据类型列表，默认值为 text/xml, application/xml, application/json。
- feign.compression.request.min-request-size：设置请求内容的最小阈值，默认值为 2048。

5.4 本章小结

本章主要讲述了 Feign 框架，Feign 框架被集成到 Spring Cloud 的 Netflix 项目中，主要作为 REST 客户端。该框架的主要优点在于，它的插件式机制可以灵活地被整合到项目中。Spring Cloud 对其进行了封装，本来使用就很简单的 Feign，在 Spring Cloud 中使用更为简单。Feign 自带 Ribbon 模块，本身就具有负载均衡的能力，可以访问集群的服务。

5.2 节主要以 Feign 的使用为核心，我们讲述了 Feign 的几个重要组成部分。在 5.3 节，我们讲述了 Feign 在 Spring Cloud 中的使用。学习完本章后，读者可以深刻了解 Feign 的机制，以及其在 Spring Cloud 中所扮演的角色。

CHAPTER

6

第6章 Spring Cloud 的保护机制

本章要点

- 实际环境中的问题
- 使用 Hystrix 改造客户端
- Hystrix 的使用
- 在 Spring Cloud 中使用 Hystrix

6.1 概述

在很多系统架构中都需要考虑横向扩展、单点故障等问题，对于一个庞大的应用集群，部分服务或者机器出现问题不可避免。在出现故障时，如何减少故障的影响、保障集群的高可用，成为一个重要的课题。在微服务集群中，不管是服务器，还是客户端，都支持集群部署，本章将讲述 Spring Cloud 中所使用的集群保护框架：Hystrix。

6.1.1 实际问题

假设有一个应用程序，调用关系如图 6-1 所示。

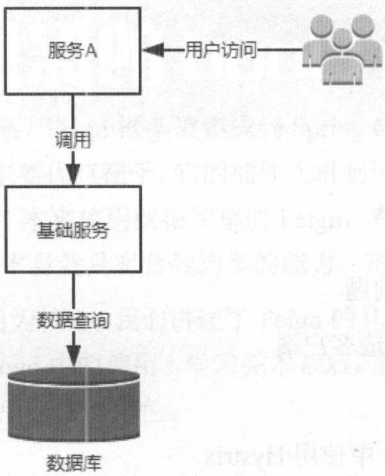


图 6-1 应用程序

图 6-1 中用户访问服务 A 模块，服务通过 Web 接口或者其他方式访问基础服务模块，基础服务模块访问数据库。

如果数据库因为某些原因变得不可用，基础服务将会得到“数据库无法访问”的信息，并且会将此信息告知服务 A 模块。在出现问题时，用户不断地请求服务 A 模块，而服务 A 模块则继续请求基础服务模块，基础服务模块仍然不停地连接有问题的数据库直到超时，大量的用户请求（包括重试的请求）会发送过来，整个应用不堪重负。

实际情况可能更加糟糕，用户的请求不停地发送给服务 A 模块，而由于数据库的原因，基础服务模块迟迟无法响应，有可能造成整个机房的网络阻塞，受害的不仅仅是该应用程序，机房中的所有服务都有可能因为网络原因而瘫痪。

6.1.2 传统的解决方式

对于前面遇到的实际问题，可以选择在连接数据库时加上超时的配置，让基础服务模块快速响应。但这仅仅算是解决了其中一种情况，在实际情况中，基础服务模块有可能出现问题，例如部分线程阻塞、进程假死等，在这些情况下，对外的服务A模块面对大量的用户与有故障的基础服务模块，仍然无法独善其身，前面所说的问题依然会出现。

笔者曾就职于某电影院售票系统供应商，在某一年的春节档期，几大互联网巨头发起了观影优惠活动，大量的用户请求涌入我们中心端的系统。由于其中某些服务节点（Tomcat）处理缓慢，很多重试、新接入的请求不断访问我们的服务。在这个时候，传说中的“人肉运维”出现了，值班的运维同事，通过手工重启Tomcat来试图缓解这种情况，然而挣扎了几个小时后，以失败告终。最终，整个集群网络阻塞，不得不停止对外服务，公司损失惨重。在这件事过后，公司方面加强了对服务节点的监控，加入了故障报告、紧急故障处理等机制，期望能减少或者避免这些问题所带来的影响。

在当今的互联网时代，面对大量的用户请求，传统或者单一的解决方式在复杂的集群面前显得有点力不从心，我们需要更优雅而且更完善的方案来解决这些问题。

6.1.3 集群容错框架Hystrix

在分布式环境中，总会有一些被依赖的服务会失效，例如像网络短暂无法访问、服务器宕机等情况。Hystrix是Netflix下的一个Java库，Spring Cloud将Hystrix整合到Netflix项目中，Hystrix通过添加延迟阈值以及容错的逻辑，来帮助我们控制分布式系统间组件的交互。Hystrix通过隔离服务间的访问点、停止它们之间的级联故障、提供可回退操作来实现容错。

例如我们前面所讲到的问题，如果数据库层面出现问题，服务A模块在访问基础模块时必定会出现超时的情况，此时可以将基础模块隔离开来，服务A在短时间内不再调用基础模块，并且快速响应用户的请求，从而保证服务A自身乃至整个集群的稳定性，这是Hystrix可以解决的问题。加入容错机制，当出现前面所说的问题时，原来的应用程序将变为图6-2所示的结构。

如图6-2所示，当前基础服务模块或者数据库不可用时，服务A将对其进行“熔断”，在一定的时间内，服务A都不会再调用基础服务，以维持本身的稳定。

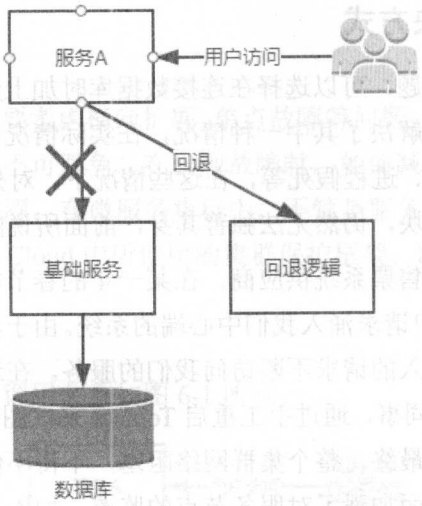


图 6-2 加入容错机制

除了服务间的依赖会导致整个集群不可用外，在其他情况下，我们同样需要集群容错。假设集群中存在 30 个服务，每个服务在 99.99%的时间内是正常运行的，计算下来，整个集群在 99.7%的时间内是正常运行的。如果该集群接受 10 亿次请求，那么将会有 300 万次请求会失败。在现实中，情况可能更加严重，每个服务有 99.99%的正常服务时间，已经是一个很乐观的数字。网络连接失败、超时，服务器硬件故障，部署引起的问题，应用程序的 bug，所有这些情况都可能会出现，不能因为单点故障而降低整个集群的可用性，容错机制变得尤为重要。

6.1.4 Hystrix 的功能

Hystrix 主要实现以下功能。

- 当所依赖的网络服务发生延迟或者失败时，对访问的客户端程序进行保护，就像前面例子中对服务 A 模块进行保护一样。
- 在分布式系统中，停止级联故障。
- 网络服务恢复正常后，可以快速恢复客户端的访问能力。
- 调用失败时执行服务回退。
- 可支持实时监控、报警和其他操作。

接下来，我们将讲述 Hystrix 的相关功能。

6.2 第一个 Hystrix 程序

先编写一个简单的 Hello World 程序，展示 Hystrix 的基本功能。注意：6.2 节与 6.3 节，Hystrix 均没有与 Spring Cloud 整合使用。

6.2.1 准备工作

使用 Spring Boot 的 spring-boot-starter-web 项目，建立一个普通的 Web 项目，发布两个测试服务用于测试，控制器的代码请见代码清单 6-1。

代码清单 6-1：codes\06\6.2\first-hystrix-server\src\main\java\org\crazyit\cloud\MyController.java

```
@RestController
public class MyController {

    @GetMapping("/normalHello")
    public String normalHello(HttpServletRequest request) {
        return "Hello World";
    }

    @GetMapping("/errorHello")
    public String errorHello(HttpServletRequest request) throws Exception {
        // 模拟需要处理 10 秒
        Thread.sleep(10000);
        return "Error Hello World";
    }
}
```

一个正常的服务，另外一个服务则需要等待 10 秒才有返回。本例的 Web 项目对应的代码目录为 codes\06\6.2\first-hystrix-server，启动类是 ServerApplication。

6.2.2 客户端使用 Hystrix

结合 Hystrix 来请求 Web 服务，可能与原来的方式不太一样。新建项目 first-hystrix-client，在 pom.xml 中加入以下依赖：

```
<dependency>
    <groupId>com.netflix.hystrix</groupId>
    <artifactId>hystrix-core</artifactId>
    <version>1.5.12</version>
```

```
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <version>1.7.25</version>
        <artifactId>slf4j-log4j12</artifactId>
    </dependency>
    <dependency>
        <groupId>commons-logging</groupId>
        <artifactId>commons-logging</artifactId>
        <version>1.2</version>
    </dependency>
    <dependency>
        <groupId>org.apache.httpcomponents</groupId>
        <artifactId>httpclient</artifactId>
        <version>4.5.2</version>
    </dependency>
```

本书 Spring Cloud 所使用的 Hystrix 的版本为 1.5.12，我们也使用与其一致的版本。客户端项目除了要使用 Hystrix 外，还会使用 HttpClient 模块访问 Web 服务，因此要加入相应的依赖。新建一个命令类，实现请见代码清单 6-2。

代码清单 6-2：codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\HelloCommand.java

```
public class HelloCommand extends HystrixCommand<String> {

    private String url;

    CloseableHttpClient httpclient;

    public HelloCommand(String url) {
        // 调用父类的构造器，设置命令组的 key，默认用来作为线程池的 key
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        // 创建 HttpClient 客户端
        this.httpclient = HttpClients.createDefault();
        this.url = url;
    }

    protected String run() throws Exception {
        try {
            // 调用 GET 方法请求服务
            HttpGet httpget = new HttpGet(url);
```

```
            // 得到服务响应
            HttpResponse response = httpclient.execute(httpget);
            // 解析并返回命令执行结果
            return EntityUtils.toString(response.getEntity());
        } catch (Exception e) {
            e.printStackTrace();
        }
        return "";
    }
}
```

新建运行类，执行 HelloCommand，如代码清单 6-3 所示。

代码清单 6-3：codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\HelloMain.java

```
public class HelloMain {

    public static void main(String[] args) {
        // 请求正常的服务
        String normalUrl = "http://localhost:8080/normalHello";
        HelloCommand command = new HelloCommand(normalUrl);
        String result = command.execute();
        System.out.println("请求正常的服务，结果：" + result);
    }
}
```

正常情况下，直接调用 HttpClient 的 API 来请求 Web 服务，而前面的命令类与运行类则通过命令来执行调用的工作。在命令类 HelloCommand 中，实现了父类的 run 方法，使用 HttpClient 调用服务的过程，都放到了该方法中。运行 HelloMain 类，可以看到，结果与平常调用 Web 服务无异。接下来，测试使用 Hystrix 的情况下调用有问题的服务。

6.2.3 调用错误服务

假设我们所调用的 Hello 服务发生故障，导致无法正常访问，那么对于客户端来说，如何自保呢？本例将调用延时的服务，为客户端设置回退方法。修改 HelloCommand 类，加入回退方法，请见代码清单 6-4。

代码清单 6-4：codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\HelloCommand.java

```
protected String getFallback() {
    System.out.println("执行 HelloCommand 的回退方法");
```

```
        return "error";
    }
```

在运行类中，调用发生故障的服务，请见代码清单 6-5。

代码清单 6-5：codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\HelloErrorMain.java

```
public class HelloErrorMain {

    public static void main(String[] args) {
        // 请求异常的服务
        String normalUrl = "http://localhost:8080/errorHello";
        HelloCommand command = new HelloCommand(normalUrl);
        String result = command.execute();
        System.out.println("请求异常的服务，结果：" + result);
    }
}
```

运行 HelloErrorMain 类，输出如下：

```
执行 HelloCommand 的回退方法
请求异常的服务，结果：error
```

根据结果可知，回退方法被执行。本例中调用的 errorHello 服务，会阻塞 10 秒才有返回。默认情况下，如果调用的 Web 服务无法在 1 秒内完成，那么将会触发回退。

回退更像是一个备胎，当请求的服务无法正常返回时，就调用该“备胎”的实现。这样做可以很好地保护客户端，服务端所提供的服务受网络等条件的制约，如果有服务真的需要 10 秒才能返回结果，而客户端又没有容错机制，后果就是，客户端将一直等待返回，直到网络超时或者服务有响应，而外界会一直不停地发送请求给客户端，最终导致的结果就是，客户端因请求过多而瘫痪。

6.2.4 Hystrix 的运作流程

在前面的例子中，使用 Hystrix 时仅仅创建命令并予以执行。看似简单，实际上，Hystrix 有一套较为复杂的执行逻辑，为了能让大家大致了解该执行过程，笔者将整个流程进行了简化。Hystrix 的运作流程请见图 6-3。

简单说明一下运作流程。

- **第一步**：在命令开始执行时，会做一些准备工作，例如为命令创建相应的线程池（后面章节讲述）等。

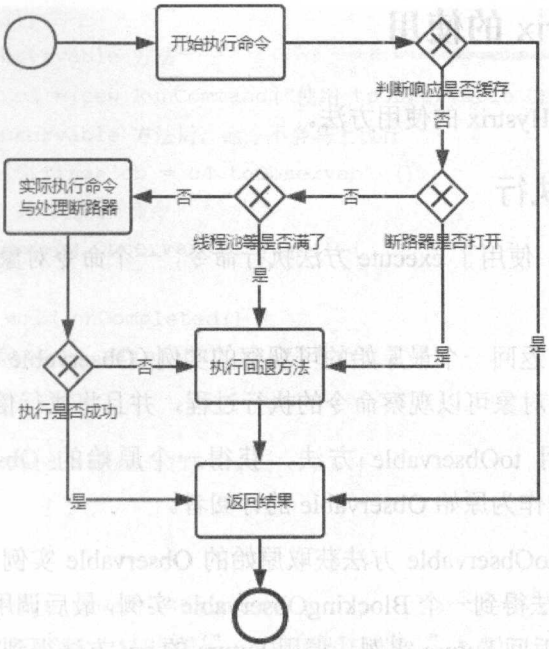


图 6-3 Hystrix 的运作流程图

- **第二步**：判断是否打开了缓存，打开了缓存就直接查找缓存并返回结果。
- **第三步**：判断断路器是否打开，如果打开了，就表示链路不可用，直接执行回退方法。结合本章开头的例子，可理解为基础服务模块不可用，服务 A 模块直接执行回退，响应用户请求。
- **第四步**：判断线程池、信号量（计数器）等条件，例如像线程池超负荷，则执行回退方法，否则，就去执行命令的内容（例如前面例子中的调用服务）。
- **第五步**：执行命令，计算是否要对断路器进行处理，执行完成后，如果满足一定条件，则需要开启断路器。如果执行成功，则返回结果，反之则执行回退。

整个流程最主要的地方在于断路器是否被打开，后面会讲解断路器的相关内容。我们的客户端在使用 Hystrix 时，表面上只是创建了一个命令来执行，实际上 Hystrix 已经为客户端添加了几层保护。

图 6-3 所示的流程图对 Hystrix 的运作流程做了最简单的描述，对于部分细节，在此不进行讲述，读者大致了解运作流程即可。

6.3 Hystrix 的使用

本节将详细讲述 Hystrix 的使用方法。

6.3.1 命令执行

在前面的例子中，使用了 execute 方法执行命令，一个命令对象可以使用以下方法来执行命令。

- toObservable：返回一个最原始的可观察的实例（Observable），Observable 是 RxJava 的类，使用该对象可以观察命令的执行过程，并且将执行信息传递给订阅者。
- observe：调用 toObservable 方法，获得一个原始的 Observable 实例后，使用 ReplaySubject 作为原始 Observable 的订阅者。
- queue：通过 toObservable 方法获取原始的 Observable 实例，再调用 Observable 的 toBlocking 方法得到一个 BlockingObservable 实例，最后调用 BlockingObservable 的 toFuture 方法返回 Future 实例，调用 Future 的 get 方法得到执行结果。
- execute：调用 queue 的 get 方法返回命令的执行结果，该方法同步执行。

以上 4 个方法，除 execute 方法外，其他方法均为异步执行。observe 与 toObservable 方法的区别在于，toObservable 被调用后，命令不会立即执行，只有当返回的 Observable 实例被订阅后，才会真正执行命令。而在 observe 方法的实现中，会调用 toObservable 得到 Observable 实例，再对其进行订阅，因此调用 observe 方法后会立即执行命令（异步）。代码清单 6-6 使用了 4 个执行命令的方法。

代码清单 6-6：codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\run\RunTest.java

```
public class RunTest {

    public static void main(String[] args) throws Exception {
        // 使用 execute 方法
        RunCommand c1 = new RunCommand("使用 execute 方法执行命令");
        c1.execute();
        // 使用 queue 方法
        RunCommand c2 = new RunCommand("使用 queue 方法执行命令");
        c2.queue();
        // 使用 observe 方法
        RunCommand c3 = new RunCommand("使用 observe 方法执行命令");
```

```
        c3.observe();
        // 使用 toObservable 方法
        RunCommand c4 = new RunCommand("使用 toObservable 方法执行命令");
        // 调用 toObservable 方法后，命令不会马上执行
        Observable<String> ob = c4.toObservable();
        // 进行订阅，此时会执行命令
        ob.subscribe(new Observer<String>() {

            public void onCompleted() {
                System.out.println("    命令执行完成");
            }

            public void onError(Throwable e) {

            }

            public void onNext(String t) {
                System.out.println("    命令执行结果: " + t);
            }

        });
        Thread.sleep(100);
    }

    // 测试命令
    static class RunCommand extends HystrixCommand<String> {

        String msg;

        public RunCommand(String msg) {
            super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
            this.msg = msg;
        }

        protected String run() throws Exception {
            System.out.println(msg);
            return "success";
        }
    }
}
```

对于 4 个执行命令的方法，读者需要知道 toObservable 与 observe 方法的区别，这两个方法将会在 Spring Cloud 中使用。

6.3.2 属性配置

使用 Hystrix 时，可以为命令设置属性，以下的代码片断为一个命令设置了执行的超时时间：

```
public MyCommand(boolean isTimeout) {
    super(
        Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"))
        .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
            .withExecutionTimeoutInMilliseconds(500))
    );
}
```

以上的配置仅对该命令生效，设置了命令的超时时间为 500 毫秒。该配置项的默认值为 1 秒，如果想对全局生效，可以使用以下代码片断：

```
ConfigurationManager
    .getConfigInstance()
    .setProperty(
    "hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds",
    500);
```

以上代码片断同样设置了命令超时时间为 500 毫秒，但对全局有效。除了超时的配置外，还需要了解一下命令的相关名称，可以为命令设置以下名称。

- 命令组名称（GroupKey）：必须提供命令组名称，默认情况下，全局维护的线程池 Map 以该值作为 key，该 Map 的 value 为执行命令的线程池。
- 命令名称（CommandKey）：可选参数。
- 线程池名称（ThreadPoolKey）：指定了线程的 key 后，全局维护的线程池 Map 将以该值作为 key。

以下的代码片断分别设置以上的 3 个 Key：

```
public RunCommand(String msg) {
    super(
        Setter.withGroupKey(
            HystrixCommandGroupKey.Factory.asKey("group-key"))
```

```
            .andCommandKey(HystrixCommandKey.Factory.asKey("command-key"))
            .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("pool-key"))

        );
    }
```

Hystrix 的配置众多，后面章节的案例中会涉及部分配置。读者如果想了解更多配置，可到以下地址查看 https://github.com/Netflix/Hystrix/wiki/Configuration。

6.3.3 回退

根据 6.2 节的流程图可知，至少会有 3 种情况触发回退（fallback）：

- 断路器被打开。
- 线程池、队列、信号量满载。
- 实际执行命令失败。

在命令中实现父类（HystrixCommand）的 getFallback()方法，即可实现回退。当以上情况发生时，将会执行回退方法。在前面的例子中已经展示了“执行命令失败”的回退，下面测试一下断路器被打开时的回退，详情请见代码清单 6-7。

代码清单 6-7：codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\fallback\FallbackTest.java

```
public class FallbackTest {

    public static void main(String[] args) {
        // 断路器被强制打开
        ConfigurationManager.getConfigInstance().setProperty(
                "hystrix.command.default.circuitBreaker.forceOpen", "true");
        FallbackCommand c = new FallbackCommand();
        c.execute();
        // 创建第二个命令，断路器关闭
        ConfigurationManager.getConfigInstance().setProperty(
                "hystrix.command.default.circuitBreaker.forceOpen", "false");
        FallbackCommand c2 = new FallbackCommand();
        c2.execute();
    }

    static class FallbackCommand extends HystrixCommand<String> {
        public FallbackCommand() {
```

```
            super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        }

        /**
         * 断路器被强制打开，该方法不会执行
         */
        protected String run() throws Exception {
            System.out.println("命令执行");
            return "";
        }

        /**
         * 回退方法，断路器打开后会执行回退
         */
        protected String getFallback() {
            System.out.println("执行回退方法");
            return "fallback";
        }
    }
}
```

如果让断路器打开，需要符合一定的条件。本例为了简单起见，在代码清单中使用了配置管理类（ConfigurationManager）将断路器强制打开与关闭。在打开断路器后，FallbackCommand 总会执行回退（getFallback）方法将断路器关闭，命令执行正常。如果断路器被打开，而命令中没有提供回退方法，将抛出以下异常：

```
    com.netflix.hystrix.exception.HystrixRuntimeException: FallbackCommand
short-circuited and no fallback available.
```

另外，需要注意的是，命令执行后，不管是否会触发回退，都会去计算整个链路的健康状况，根据健康状况来判断是否要打开断路器。如果命令仅仅失败了一次，是不足以打开断路器的，关于断路器的逻辑将在后面章节讲述。

➤➤ 6.3.4 回退的模式

Hystrix 的回退机制比较灵活，你可以在 A 命令的回退方法中执行 B 命令，如果 B 命令也执行失败，同样也会触发 B 命令的回退，这样就形成一种链式的命令执行，例如以下代码片断：

```
static class CommandA extends HystrixCommand<String> {
    ……省略其他代码
    protected String run() throws Exception {
        throw new RuntimeException();
    }

    protected String getFallback() {
        return new CommandB().execute();
    }
}
```

还有其他较为复杂的例子，例如银行转账。假设一个转账命令包含调用 A 银行扣款、B 银行加款两个命令，其中一个命令失败后，执行转账命令的回退，如图 6-4 所示。

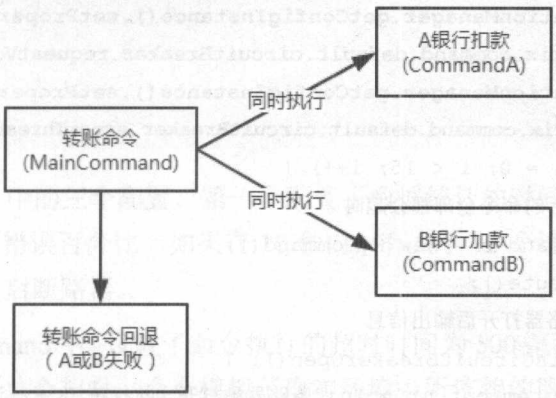


图 6-4 多命令回退

要做到图 6-4 所示的多命令只执行一次回退的效果，CommandA 与 CommandB 不能有回退方法。如果 CommandA 命令执行失败，并且该命令有回退方法，此时将不会执行 MainCommand 的回退方法。除了上面所提到的链式的回退以及多命令回退，读者还可以根据实际情况来设计回退。

6.3.5 断路器开启

断路器一旦开启，就会直接调用回退方法，不再执行命令，而且也不会更新链路的健康状况。断路器的开启要满足两个条件：

- 整个链路达到一定阈值，默认情况下，10 秒内产生超过 20 次请求，则符合第一个条件。

- 满足第一个条件的情况下，如果请求的错误百分比大于阈值，则会打开断路器，默认为50%。

Hystrix的逻辑是先判断是否满足第一个条件，再判断是否满足第二个条件。如果两个条件都满足，则开启断路器。断路器开启的测试代码请见代码清单6-8。

代码清单6-8：codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\breaker\OpenTest.java

```
public class OpenTest {

    public static void main(String[] args) throws Exception {
        // 10秒内有10个请求，则符合第一个条件
        ConfigurationManager.getConfigInstance().setProperty(
            "hystrix.command.default.metrics.rollingStats.timeInMilliseconds", 10000);
        ConfigurationManager.getConfigInstance().setProperty(
            "hystrix.command.default.circuitBreaker.requestVolumeThreshold", 10);
        ConfigurationManager.getConfigInstance().setProperty(
            "hystrix.command.default.circuitBreaker.errorThresholdPercentage", 50);
        for(int i = 0; i < 15; i++) {
            // 执行的命令全部都会超时
            MyCommand c = new MyCommand();
            c.execute();
            // 断路器打开后输出信息
            if(c.isCircuitBreakerOpen()) {
                System.out.println("断路器被打开，执行第 " + (i + 1) + " 个命令");
            }
        }
    }

    /**
     * 模拟超时的命令
     * @author 杨恩雄
     *
     */
    static class MyCommand extends HystrixCommand<String> {
        // 设置超时的时间为500毫秒
        public MyCommand() {
            super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey
("ExampleGroup"))
```

```
                .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
                        .withExecutionTimeoutInMilliseconds(500))
                );
        }

        protected String run() throws Exception {
            // 模拟处理超时
            Thread.sleep(800);
            return "";
        }

        @Override
        protected String getFallback() {
            return "";
        }
    }
}
```

注意代码清单 6-8 中的三个配置，第一个配置了数据统计的时间，第二个配置了请求的阈值，第三个配置了错误百分比。如果在 10 秒内，有大于 10 个请求发生，并且请求的错误率超过 50%，则开启断路器。

在命令类 MyCommand 中，设置了命令执行的超时时间为 500 毫秒，命令执行需要 800 毫秒，换言之，该命令总会超时，命令模拟了现实环境中所依赖的服务瘫痪（超时响应）的情况。

在运行类中，循环 15 次执行命令，调用 isCircuitBreakerOpen 方法，如果断路器打开，则输出信息。运行代码清单 6-8 所示的 OpenTest 类，输出如下：

```
断路器被打开，执行第 11 个命令
断路器被打开，执行第 12 个命令
断路器被打开，执行第 13 个命令
断路器被打开，执行第 14 个命令
断路器被打开，执行第 15 个命令
```

根据结果可知，前面执行的 10 个命令没有开启断路器，而到了第 11 个命令，断路器被打开，命令不再执行。

6.3.6 断路器关闭

断路器打开后，在一段时间内，命令不会再执行（一直触发回退），这段时间我们称作“休眠期”。休眠期的默认值为5秒，休眠期结束后，Hystrix会尝试性地执行一次命令，此时断路器的状态不是开启，也不是关闭，而是一个半开的状态，如果这一次命令执行成功，则会关闭断路器并清空链路的健康信息；如果执行失败，断路器会继续保持打开的状态。断路器的打开与关闭测试，请见代码清单6-9。

代码清单6-9：codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\breaker\CloseTest.java

```
public class CloseTest {

    public static void main(String[] args) throws Exception {
        // 10秒内有3个请求就满足第一个开启断路器的条件
        ConfigurationManager.getConfigInstance().setProperty(
            "hystrix.command.default.metrics.rollingStats.timeInMilliseconds", 10000);
        ConfigurationManager.getConfigInstance().setProperty(
            "hystrix.command.default.circuitBreaker.requestVolumeThreshold", 3);
        // 请求的失败率，默认值为50%
        ConfigurationManager.getConfigInstance().setProperty(
            "hystrix.command.default.circuitBreaker.errorThresholdPercentage", 50);
        // 设置休眠期，断路器打开后，这段时间不会再执行命令，默认值为5秒，此处设置为3秒
        ConfigurationManager.getConfigInstance().setProperty(
            "hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds", 3000);
        // 该值决定是否执行超时
        boolean isTimeout = true;
        for(int i = 0; i < 10; i++) {
            // 执行的命令全部都会超时
            MyCommand c = new MyCommand(isTimeout);
            c.execute();
            // 输出健康状态等信息
            HealthCounts hc = c.getMetrics().getHealthCounts();
            System.out.println("断路器状态：" + c.isCircuitBreakerOpen() +
                    "，请求总数：" + hc.getTotalRequests());
            if(c.isCircuitBreakerOpen()) {
                // 断路器打开，让下一次循环成功执行命令
                isTimeout = false;
                System.out.println("=====  断路器打开了，等待休眠期结束  =====");
```

```
                // 休眠期会在3秒后结束，此处等待4秒，确保休眠期结束
                Thread.sleep(4000);
            }
        }
    }

    /**
     * 模拟超时的命令
     * @author 杨恩雄
     *
     */
    static class MyCommand extends HystrixCommand<String> {

        private boolean isTimeout;

        // 设置超时的时间为500毫秒
        public MyCommand(boolean isTimeout) {
            super(
                Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey
("ExampleGroup"))
                .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
                        .withExecutionTimeoutInMilliseconds(500))
                );
            this.isTimeout = isTimeout;
        }

        protected String run() throws Exception {
            // 让外部决定是否超时
            if(isTimeout) {
                // 模拟处理超时
                Thread.sleep(800);
            } else {
                Thread.sleep(200);
            }
            return "";
        }

        @Override
        protected String getFallback() {
```

```
            return "";
        }
    }
}
```

代码清单中配置了休眠期为 3 秒，循环 10 次，创建 10 个命令并执行。在执行完第 4 个命令后，断路器会被打开，此时我们等待休眠期结束，让下一次循环的命令执行成功。

代码清单中使用了一个布尔值来决定是否执行成功，第 5 次命令会执行成功，此时断路器将会被关闭，剩下的命令全部都可以正常执行。在循环体中，使用了 HealthCounts 对象，该对象用于记录链路的健康信息。如果断路器关闭（链路恢复健康），HealthCounts 里面的健康信息将会被重置。运行代码清单 6-9，效果如下：

```
断路器状态：false，请求总数：0
断路器状态：false，请求总数：1
断路器状态：false，请求总数：2
断路器状态：true，请求总数：3
=====  断路器打开了，等待休眠期结束   =====
断路器状态：false，请求总数：0
断路器状态：false，请求总数：1
断路器状态：false，请求总数：1
断路器状态：false，请求总数：3
断路器状态：false，请求总数：3
断路器状态：false，请求总数：5
```

6.3.7 隔离机制

命令的真正执行，除了断路器要关闭外，还需要再过一关：执行命令的线程池或者信号量是否满载。如果满载，命令就不会执行，而是直接触发回退，这样的机制，在控制命令的执行上，实现了错误的隔离。Hystrix 提供了两种隔离策略。

- THREAD：默认值，由线程池来决定命令的执行，如线程池满载，则不会执行命令。Hystrix 使用了 ThreadPoolExecutor 来控制线程池行为，线程池的默认大小为 10。
- SEMAPHORE：由信号量来决定命令的执行，当请求的并发数高于阈值时，就不再执行命令。相对于线程策略，信号量策略开销更小，但是该策略不支持超时以及异步，除非对调用的服务有足够的信任，否则不建议使用该策略进行隔离。

接下来，使用代码测试线程隔离与信号隔离两个策略，编写共用的命令类，请见代码清单6-10。

代码清单6-10：

codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\isolation\MyCommand.java

```
public class MyCommand extends HystrixCommand<String> {
    int index;
    public MyCommand(int index) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory
                .asKey("ExampleGroup")));
        this.index = index;
    }

    protected String run() throws Exception {
        Thread.sleep(500);
        System.out.println("执行方法，当前索引：" + index);
        return "";
    }

    @Override
    protected String getFallback() {
        System.out.println("执行 fallback，当前索引：" + index);
        return "";
    }
}
```

编写线程隔离的运行方法，请见代码清单6-11。

代码清单6-11：codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\isolation\ThreadIso.java

```
public class ThreadIso {

    public static void main(String[] args) throws Exception {
        // 配置线程池大小为3
        ConfigurationManager.getConfigInstance().setProperty(
                "hystrix.threadpool.default.coreSize", 3);
        for(int i = 0; i < 6; i++) {
            MyCommand c = new MyCommand(i);
            c.queue();
        }
```

```
            Thread.sleep(5000);
        }
    }
```

在使用线程隔离策略的运行类中，配置了线程池大小为3，进行6次循环，意味着有3次命令将会触发回退，运行后可看到效果。代码清单6-12测试信号量隔离策略。

代码清单6-12：

codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\isolation\SemaphoreIso.java

```
public class SemaphoreIso {

    public static void main(String[] args) throws Exception {
        // 配置使用信号量的策略进行隔离
        ConfigurationManager.getConfigInstance().setProperty(
                "hystrix.command.default.execution.isolation.strategy",
                ExecutionIsolationStrategy.SEMAPHORE);
        // 设置最大并发数，默认值为10，本例设置为2
        ConfigurationManager
                .getConfigInstance()
                .setProperty("hystrix.command.default.execution.isolation.semap
hore.maxConcurrentRequests",
                        2);
        // 设置执行回退方法的最大并发，默认值为10，本例设置为20
        ConfigurationManager
                .getConfigInstance()
                .setProperty("hystrix.command.default.fallback.isolation.semaph
ore.maxConcurrentRequests",
                        20);
        for (int i = 0; i < 6; i++) {
            final int index = i;
            Thread t = new Thread() {
                public void run() {
                    MyCommand c = new MyCommand(index);
                    c.execute();
                }
            };
            t.start();
        }
        Thread.sleep(5000);
    }
}
```

注意代码中的3个配置项，指定使用信号量作为隔离策略，分别设置了命令执行的最

大并发数，以及执行回退的最大并发数。运行代码清单 6-12，最终只有两个命令正常执行，其余命令都会触发回退，可见信号量隔离的相关配置生效。

线程（线程池）与信号量两种隔离策略各有优缺点，如果对于所调用的服务有足够的信任，可以使用信号量策略，以减少系统开销。

6.3.8 合并请求

根据前面小节的介绍可知，默认情况下，会为命令分配线程池来执行命令实例，线程池会消耗一定的性能。对于一些同类型的请求（URL 相同，参数不同），Hystrix 提供了合并请求的功能，在一次请求的过程中，可以将一个时间段内的相同请求（参数不同），收集到同一个命令中执行，这样就节省了线程的开销，减少了网络连接，从而提升了执行的性能。这个功能有点像数据库的批处理功能。

实现合并请求的功能，至少包含以下 3 个条件：

- 需要有一个执行请求的命令，将全部参数进行整理，然后调用外部服务。
- 需要有一个合并处理器，用于收集请求，以及处理结果。
- 外部接口提供支持，例如外部的服务提供了/person/{personName}的服务用于查找一个 Person，如果合并请求，外部还需要提供一个/persons 的服务，用于查找多个 Person。

接下来，实现一个简单的查找逻辑。假设有以下场景：客户端多次调用查找单个 Person 的 Web 服务，而服务端提供了一个新的服务，可以传入多个名字，查找并返回多个 Person 实例，此时，可以考虑使用合并请求。编写一个命令类，用于收集请求参数以及调用服务，请见代码清单 6-13。

代码清单 6-13：

codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\collapse\CollapseTest.java

```
static class CollapserCommand extends HystrixCommand<Map<String, Person>> {
    // 请求集合，第一个类型是单个请求返回的数据类型，第二是请求参数的类型
    Collection<CollapsedRequest<Person, String>> requests;

    private CollapserCommand(
            Collection<CollapsedRequest<Person, String>> requests) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory
                .asKey("ExampleGroup")));
        this.requests = requests;
    }
```

```
    @Override
    protected Map<String, Person> run() throws Exception {
        System.out.println("收集参数后执行命令，参数数量：" + requests.size());
        // 处理参数
        List<String> personNames = new ArrayList<String>();
        for(CollapsedRequest<Person, String> request : requests) {
            personNames.add(request.getArgument());
        }
        // 调用服务（此处模拟调用），根据名称获取 Person 的 Map
        Map<String, Person> result = callService(personNames);
        return result;
    }

    // 模拟服务返回
    private Map<String, Person> callService(List<String> personNames) {
        Map<String, Person> result = new HashMap<String, Person>();
        for(String personName : personNames) {
            Person p = new Person();
            p.id = UUID.randomUUID().toString();
            p.name = personName;
            p.age = new Random().nextInt(30);
            result.put(personName, p);
        }
        return result;
    }
}

static class Person {
    String id;
    String name;
    Integer age;

    public String toString() {
        // TODO Auto-generated method stub
        return "id: " + id + ", name: " + name + ", age: " + age;
    }
}
```

在命令类 CollapserCommand 中，维护着一个 CollapsedRequest 集合，一个 CollapsedRequest

实例表示一个请求，该类指定的第一个类型为“单请求返回的类型”，第二个类型为请求的参数类型。例如，CollapsedRequest<Person, String>，表示单次请求将会以 String 作为参数，返回一个 Person 实例。代码清单中的粗体部分模拟调用查询多个 Person 的服务。

接下来，编写合并处理器，将请求进行合并，请见代码清单 6-14。

代码清单 6-14：

codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\collapse\CollapseTest.java

```
/**
 * 合并处理器
 * 第一个类型为批处理返回的结果类型
 * 第二个为单请求返回的结果类型
 * 第三个是请求参数类型
 * @author 杨恩雄
 */
static class MyHystrixCollapser extends
        HystrixCollapser<Map<String, Person>, Person, String> {

    String personName;

    public MyHystrixCollapser(String personName) {
        this.personName = personName;
    }

    @Override
    public String getRequestArgument() {
        return personName;
    }

    @Override
    protected HystrixCommand<Map<String, Person>> createCommand(
            Collection<CollapsedRequest<Person, String>> requests) {
        return new CollapserCommand(requests);
    }

    @Override
    protected void mapResponseToRequests(Map<String, Person> batchResponse,
            Collection<CollapsedRequest<Person, String>> requests) {
        // 让结果与请求进行关联
```

```
        for (CollapsedRequest<Person, String> request : requests) {
            // 获取单个响应返回的结果
            Person singleResult = batchResponse.get(request.getArgument());
            // 关联到请求中
            request.setResponse(singleResult);
        }
    }
}
```

合并处理器中实现了父类的 3 个方法，getRequestArgument 用于返回请求的参数，本例需要根据名称查询 Person，因此该参数为字符串；createCommand 返回实际执行的命令（查找多个 Person 的批处理命令）；mapResponseToRequests 方法将会在返回结果后执行，可在该方法中设置结果与请求之间的关联。

接下来，编写运行类，请见代码清单 6-15。

代码清单 6-15：

codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\collapse\CollapseTest.java

```
public static void main(String[] args) throws Exception {
    // 收集 1 秒内发生的请求，合并为一个命令执行
    ConfigurationManager.getConfigInstance().setProperty(
            "hystrix.collapser.default.timerDelayInMilliseconds", 1000);
    // 请求上下文
    HystrixRequestContext context = HystrixRequestContext
            .initializeContext();
    // 创建请求合并处理器
    MyHystrixCollapser c1 = new MyHystrixCollapser("Angus");
    MyHystrixCollapser c2 = new MyHystrixCollapser("Crazyit");
    MyHystrixCollapser c3 = new MyHystrixCollapser("Sune");
    MyHystrixCollapser c4 = new MyHystrixCollapser("Paris");
    // 异步执行
    Future<Person> f1 = c1.queue();
    Future<Person> f2 = c2.queue();
    Future<Person> f3 = c3.queue();
    Future<Person> f4 = c4.queue();
    System.out.println(f1.get());
    System.out.println(f2.get());
    System.out.println(f3.get());
    System.out.println(f4.get());
    context.shutdown();
}
```

注意粗体部分，设置了“时间段”，在1秒内执行的请求将会被合并到一起执行，该“时间段”的默认值为10毫秒。运行代码清单6-15，可以看到输出如下：

```
收集参数后执行命令，参数数量：4
id: 57821cac-d925-4a8e-9cd4-067c8171a626, name: Angus, age: 27
id: 9f984af6-0e7d-41b0-96f9-941863c3e093, name: Crazyit, age: 2
id: c8483f21-9382-436a-b01b-3ae1c8ecfb68, name: Sune, age: 24
id: 5514a5e4-8af6-4730-ab5c-080affcecc2e, name: Paris, age: 22
```

根据结果可知，创建了4个合并处理器，最终只执行了1次命令。

注意：

虽然合并请求后只执行了一个命令，只启动了一个线程，只进行了一次网络请求，但是在收集请求、合并请求、处理结果的过程中仍然会耗费一定的时间。总的来说，一般情况下，合并请求进行批处理，比发送多个请求快，对于一些服务的URL相同、参数不同的请求，笔者推荐使用合并请求的功能。

6.3.9 请求缓存

Hystrix支持缓存功能，如果在一次请求的过程中，多个地方调用同一个接口，可以考虑使用缓存。缓存打开后，下一次的命令不会执行，直接到缓存中获取响应并返回，具体可参见6.2.4节介绍的运作流程。开启缓存较为简单，在命令中重写父类的getCacheKey即可。代码清单6-16测试开启缓存和清空缓存。

代码清单6-16：codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\cache\CacheMain.java

```
public class CacheMain {

    public static void main(String[] args) {
        // 初始化请求上下文
        HystrixRequestContext context = HystrixRequestContext.initializeContext();
        // 请求正常的服务
        String key = "cache-key";
        MyCommand c1 = new MyCommand(key);
        MyCommand c2 = new MyCommand(key);
        MyCommand c3 = new MyCommand(key);
        // 输出结果
        System.out.println(c1.execute() + "c1 是否读取缓存: " +
            c1.isResponseFromCache());
```

```
        System.out.println(c2.execute() + "c2 是否读取缓存: " +
            c2.isResponseFromCache());
        System.out.println(c3.execute() + "c3 是否读取缓存: " +
            c3.isResponseFromCache());
        // 获取缓存实例
        HystrixRequestCache cache = HystrixRequestCache.getInstance(
                HystrixCommandKey.Factory.asKey("MyCommandKey"),
                HystrixConcurrencyStrategyDefault.getInstance());
        // 清空缓存
        cache.clear(key);
        // 重新执行命令
        MyCommand c4 = new MyCommand(key);
        System.out.println(c4.execute() + "c4 是否读取缓存: " +
            c4.isResponseFromCache());
        context.shutdown();
    }

    static class MyCommand extends HystrixCommand<String> {

        private String key;

        public MyCommand(String key) {
            super(
                    Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey
                        ("ExampleGroup"))
                    .andCommandKey(HystrixCommandKey.Factory.asKey("MyCommandKey"))
                    );
            this.key = key;
        }

        protected String run() throws Exception {
            System.out.println("执行命令");
            return "";
        }

        @Override
        protected String getCacheKey() {
            return this.key;
        }
    }
}
```

在代码清单的运行类中，使用了同一个缓存的key来创建命令实例。注意粗体代码，使用HystrixRequestCache来清空缓存。获取HystrixRequestCache实例需要传入CommandKey，因此在命令的构造器中，也设置了CommandKey。运行代码清单6-16，输出结果如下：

```
执行命令
c1 是否读取缓存: false
c2 是否读取缓存: true
c3 是否读取缓存: true
执行命令
c4 是否读取缓存: false
```

根据输出结果可知，命令实际上只执行了两次，c2与c3这两个命令执行时都读取了缓存，而c4在执行前清空了缓存。

注意：

合并请求、请求缓存，在一次请求的过程中才能实现，因此需要先初始化请求上下文。

Hystrix的使用到此结束，下面将讲解在Spring Cloud中如何使用Hystrix。Spring Cloud对Hystrix进行了封装，我们只要懂得Hystrix的机制或者原理，在使用Spring Cloud的Hystrix注解时，就会变得非常轻松，也可以大概知道Spring Cloud帮我们做了哪些工作。

6.4 在Spring Cloud中使用Hystrix

Hystrix主要用于保护调用服务的一方，如果被调用的服务发生故障，符合一定条件，就开启断路器，对调用的程序进行隔离。在开始讲述本节的内容之前，先准备测试项目，本节中的例子所使用的项目如下所述。

- spring-hystrix-server：Eureka服务器，端口为8761，代码目录为codes\06\6.4\spring-hystrix-server。
- spring-hystrix-provider：服务提供者，本例只需要启动一个实例，端口为8080，默认提供/person/{personId}服务。根据personId参数返回一个Person实例，另外再提供一个/hello服务，返回普通的字符串。代码目录为codes\06\6.4\spring-hystrix-provider。

- spring-hystrix-invoker：服务调用者，端口为 9000，代码目录为 codes\06\6.4\spring-hystrix-invoker。

6.4.1 整合 Hystrix

为服务调用者（spring-hystrix-invoker）项目添加依赖，添加的依赖如下：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-ribbon</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

在服务调用者的应用启动类中，加入启用断路器的注解，请参见以下代码片断：

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
public class InvokerApplication {

    @LoadBalanced
    @Bean
    public RestTemplate getRestTemplate() {
        return new RestTemplate();
    }

    public static void main(String[] args) {
        SpringApplication.run(InvokerApplication.class, args);
    }
}
```

新建服务类，在服务方法中调用服务，请见代码清单6-17。

代码清单6-17：codes\06\6.4\spring-hystrix-invoker\src\main\java\org\crazyit\cloud\PersonService.java

```
@Component
public class PersonService {

    @Autowired
    private RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "getPersonFallback")
    public Person getPerson(Integer id) {
        // 使用 RestTemplate 调用 Eureka 服务
        Person p = restTemplate.getForObject(
                "http://spring-hystrix-provider/person/{personId}",
                Person.class, id);
        return p;
    }

    /**
     * 回退方法，返回一个默认的 Person
     */
    public Person getPersonFallback(Integer id) {
        Person p = new Person();
        p.setId(0);
        p.setName("Crazyit");
        p.setAge(-1);
        p.setMessage("request error");
        return p;
    }
}
```

服务类中注入了RestTemplate，服务方法使用@HystrixCommand注解进行修饰，并且配置了回退方法。@HystrixCommand注解由Hystrix的javanica项目提供，该项目主要是为了简化Hystrix的使用。被@HystrixCommand修饰的方法，Hystrix（javanica）会使用AspectJ对其进行代理，Spring会将相关的类转换为Bean放到容器中，在Spring Cloud中，我们无须过多关心Hystrix的命令管理。

接下来，编写控制器，调用服务类的方法，请见代码清单6-18。

代码清单 6-18：

codes\06\6.4\spring-hystrix-invoker\src\main\java\org\crazyit\cloud\InvokerController.java

```
@RestController
@Configuration
public class InvokerController {

    @Autowired
    private PersonService personService;

    @RequestMapping(value = "/router/{personId}", method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON_VALUE)
    public Person router(@PathVariable Integer personId) {
        Person p = personService.getPerson(personId);
        return p;
    }
}
```

控制器的实现较为简单，直接注入 PersonService，然后调用方法即可，按以下步骤启动集群：

- 启动 spring-hystrix-server，本例中配置端口为 8761。
- 启动 spring-hystrix-provider，启动一个实例，端口为 8080。
- 启动 spring-hystrix-invoker，端口为 9000。

打开浏览器访问 http://localhost:9000/router/1，输出如下：

```
{"id":1,"name":"Crazyit","age":33,"message":"http://localhost:8080/person/1"}
```

停止服务提供者（spring-hystrix-provide），即停止 8080 端口，再访问 9000 端口的地址，输出如下：

```
{"id":0,"name":"Crazyit","age":-1,"message":"request error"}
```

根据输出可知，由于调用失败，触发了回退方法。

6.4.2 命令配置

在 Spring Cloud 中使用@HystrixCommand 来声明一个命令，命令的相关配置也可以在该注解中进行，以下的代码片断配置了几个属性：

```
/**
 * 测试配置，对3个key进行命名
 * 设置命令执行超时时间为1000毫秒
 * 设置命令执行的线程池大小为1
 */
@HystrixCommand(
        fallbackMethod="testConfigFallback", groupKey="MyGroup",
        commandKey="MyCommandKey", threadPoolKey="MyCommandPool",
        commandProperties={
                @HystrixProperty(name = "execution.isolation.thread.
                        timeoutInMilliseconds", value = "1000")
        },
        threadPoolProperties={
                @HystrixProperty(name = "coreSize",
                        value = "1")
        })
```

除了以上的几个配置外，@HystrixCommand注解还可以使用ignoreExceptions来处理异常的传播，请见以下代码片断：

```
/**
 * 声明了忽略MyException，如果方法抛出MyException，则不会触发回退
 */
@HystrixCommand(ignoreExceptions = {MyException.class},
        fallbackMethod="testExceptionFallBack")
public String testException() {
    throw new MyException();
}
```

Hystrix的命令、线程配置较多，由于篇幅所限，本小节仅简单地列举几个，读者可举一反三，按需要进行配置。

6.4.3 默认配置

对于一些默认的配置，例如命令组的key等，可以使用@DefaultProperties注解，这样就减少了@HystrixCommand注解的代码量。以下代码片断展示了如何使用@DefaultProperties：

```
@DefaultProperties(groupKey="GroupPersonKey")
public class PersonService {
```

```
    @HystrixCommand // groupkey将使用GroupPersonKey
    public String hello() {
        return "";
    }
}
```

除了定义 GroupKey 外，还支持@HystrixCommand 的其余配置，例如线程属性、命令属性等。

6.4.4 缓存注解

在 6.3 节中讲述了 Hystrix 的缓存功能，在 Spring Cloud 中，同样支持使用缓存，并且可以通过注解来实现。根据前面章节的介绍可知，缓存与合并请求功能需要先初始化请求上下文才能实现。新建一个 javax.servlet.Filter，用于创建与销毁 Hystrix 的请求上下文，请见代码清单 6-19。

代码清单 6-19：codes\06\6.4\spring-hystrix-invoker\src\main\java\org\crazyit\cloud\HystrixFilter.java

```
@WebFilter(urlPatterns = "/*", filterName = "hystrixFilter")
public class HystrixFilter implements Filter {

    public void init(FilterConfig filterConfig) throws ServletException {
    }
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        HystrixRequestContext context = HystrixRequestContext
                .initializeContext();
        try {
            chain.doFilter(request, response);
        } finally {
            context.shutdown();
        }
    }
    public void destroy() {
    }
}
```

编写服务方法，使用@CacheResult 注解进行修饰，请见代码清单 6-20。

代码清单 6-20：codes\06\6.4\spring-hystrix-invoker\src\main\java\org\crazyit\cloud\cache\CacheService.java

```
@Component
public class CacheService {

    @CacheResult
    @HystrixCommand
    public Person getPerson(Integer id) {
        System.out.println("执行 getPerson 方法");
        Person p = new Person();
        p.setId(id);
        p.setName("angus");
        return p;
    }
}
```

注意，在服务方法中，被调用一次就会进行一次控制台输出。在控制器的方法中，调用多次 getPerson 方法，控制器代码请见代码清单 6-21。

代码清单 6-21：

codes\06\6.4\spring-hystrix-invoker\src\main\java\org\crazyit\cloud\InvokerController.java

```
@RequestMapping(value = "/cache1/{personId}", method = RequestMethod.GET,
        produces = MediaType.APPLICATION_JSON_VALUE)
public Person testCacheResult(@PathVariable Integer personId) {
    // 调用多次服务
    for(int i = 0; i < 3; i++) {
        Person p = cacheService.getPerson(personId);
        System.out.println("控制器调用服务 " + i);
    }
    return new Person();
}
```

控制器中调用了多次服务方法，也就是用户发送请求后，会执行多次服务方法，启动“服务调用者”，访问以下地址 http://localhost:9000/cache1/1，控制台输出如下：

```
执行 getPerson 方法
控制器调用服务 0
控制器调用服务 1
控制器调用服务 2
```

根据输出结果可知，在一次用户请求的过程中，服务方法只执行了一次，缓存生效。

缓存的注解主要有以下 3 个。

- @CacheResult：该注解修饰方法，表示被修饰的方法返回结果将会被缓存，需要与@HystrixCommand 一起使用。
- @CacheRemove：用于修饰方法让缓存失效，需要与@CacheResult 的缓存 key 关联。
- @CacheKey：用于修饰方法参数，表示该参数作为缓存的 key。

前面的例子使用了@CacheResult 注解，下面的代码片断，@CacheResult 与@CacheRemove 一起使用：

```
@CacheResult()
@HystrixCommand(commandKey = "removeKey")
public String cacheMethod(String name) {
    return "hello";
}

@CacheRemove(commandKey = "removeKey")
@HystrixCommand
public String updateMethod(String name) {
    return "update";
}
```

以上代码片断中的 cacheMethod 方法，使用的缓存 key 为 removeKey，方法 updateMethod 被调用后，将会删除 key 为 updateMethod 的缓存。关于 3 个缓存注解更深入的使用，本小节不进行讲述，读者可以自行测试。

6.4.5 合并请求注解

在 Spring Cloud 中同样支持合并请求，在一次 HTTP 请求的过程中，收集一段时间内的相同请求，放到一个批处理命令中执行。实现合并请求，同样需要先初始化请求上下文，具体请参见 6.4.4 节中的 Filter。接下来，编写服务类，请见代码清单 6-22。

代码清单 6-22：

codes\06\6.4\spring-hystrix-invoker\src\main\java\org\crazyit\cloud\collapse\CollapseService.java

```
@Component
public class CollapseService {

    // 配置收集 1 秒内的请求
    @HystrixCollapser(batchMethod = "getPersons", collapserProperties =
```

```
        {
            @HystrixProperty(name = "timerDelayInMilliseconds", value = "1000")
        }
    )
    public Future<Person> getSinglePerson(Integer id) {
        System.out.println("执行单个获取的方法");
        return null;
    }

    @HystrixCommand
    public List<Person> getPersons(List<Integer> ids) {
        System.out.println("收集请求，参数数量：" + ids.size());
        List<Person> ps = new ArrayList<Person>();
        for (Integer id : ids) {
            Person p = new Person();
            p.setId(id);
            p.setName("crazyit");
            ps.add(p);
        }
        return ps;
    }
}
```

在代码清单中，最后真实执行的方法为 getPersons，getSinglePerson 方法使用了@HystrixCollapser 注解来修饰，会收集 1 秒内调用 getSinglePerson 的请求，放到 getPersons 方法中进行批处理。控制器中多次调用 getSinglePerson 方法，如代码清单 6-23 所示。

代码清单 6-23：

codes\06\6.4\spring-hystrix-invoker\src\main\java\org\crazyit\cloud\InvokerController.java

```
@RequestMapping(value = "/collapse", method = RequestMethod.GET,
        produces = MediaType.APPLICATION_JSON_VALUE)
public String testCollapse() throws Exception {
    // 连续执行 3 次请求
    Future<Person> f1 = collapseService.getSinglePerson(1);
    Future<Person> f2 = collapseService.getSinglePerson(2);
    Future<Person> f3 = collapseService.getSinglePerson(3);
    Person p1 = f1.get();
    Person p2 = f2.get();
    Person p3 = f3.get();
    System.out.println(p1.getId() + "---" + p1.getName());
```

```
        System.out.println(p2.getId() + "---" + p2.getName());
        System.out.println(p3.getId() + "---" + p3.getName());
        return "";
    }
```

异步执行了3次getSinglePerson方法，启动“服务调用者”，访问以下地址http://localhost:9000/collapse，控制台输出如下：

```
收集请求，参数数量：3
1---crazyit
2---crazyit
3---crazyit
```

根据输出结果可知，最终只执行了getPersons方法。相对于直接使用Hystrix，在Spring Cloud中合并请求较为简单，合并处理器已经由@HystrixCollapser注解帮我们实现，我们仅关心真正命令的执行即可。

6.4.6 Feign与Hystrix整合

Feign对Hystrix提供了支持，为“服务调用者”加入以下Feign依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

在application.yml中打开Feign的Hystrix开关，请见以下配置：

```
feign:
  hystrix:
    enabled: true
```

在应用启动类中加入Feign的开关，本小节的“服务调用者”应用启动类所使用的注解如下：

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
@ServletComponentScan
@EnableFeignClients
```

新建Feign接口，调用“服务提供者（spring-hystrix-provider）”的/hello服务，请见代

码清单 6-24。

代码清单 6-24：codes\06\6.4\spring-hystrix-invoker\src\main\java\org\crazyit\cloud\feign\HelloClient.java

```
@FeignClient(name = "spring-hystrix-provider", fallback = HelloClientFallback.
    class)
public interface HelloClient {

    @RequestMapping(method = RequestMethod.GET, value = "/hello")
    public String hello();

    @Component
    static class HelloClientFallback implements HelloClient {

        public String hello() {
            System.out.println("hello 方法的回退");
            return "error hello";
        }
    }
}
```

与普通的 Feign 客户端无异，仅仅设置了处理回退的类，回退类实现了客户端接口。为了能测试效果，修改服务器端的/hello 服务，让其有 800 毫秒的延时。根据前面章节的介绍可知，默认情况下，Hystrix 的超时时间为 1 秒，因此，还需要修改超时设置。请见代码清单 6-25，在 application.yml 中修改命令配置。

代码清单 6-25：codes\06\6.4\spring-hystrix-invoker\src\main\resources\application.yml

```
hystrix:
  command:
    HelloClient#hello():
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 500
      circuitBreaker:
        requestVolumeThreshold: 3
```

注意，如果是针对全局配置，则使用与下面类似的配置片断：

```
// 默认时间段内发生的请求数
hystrix.command.default.circuitBreaker.requestVolumeThreshold
```

```
// 超时时间
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds
```

如果针对某个客户端，使用下面的配置片断：

```
hystrix.command.CommandKey.circuitBreaker.requestVolumeThreshold
```

Feign与Hystrix整合使用时，会自动帮我们生成CommandKey，格式为：Feign客户端接口名#方法名()。例如本例中的客户端为HelloClient，方法为hello，生成的CommandKey为HelloClient#hello()。而默认情况下，生成的GroupKey为@FeignClient注解的name属性。

在以上的配置中，我们针对hello方法设置了超时时间为500毫秒，而/hello服务超时时间为800毫秒，换言之，hello方法总会超时。另外，如果请求超过3次并且失败率超过50%，断路器将被打开。编写控制器，调用hello服务，并查看断路器的情况，请见代码清单6-26。

代码清单6-26：

codes\06\6.4\spring-hystrix-invoker\src\main\java\org\crazyit\cloud\feign\HelloController.java

```
@RestController
public class HelloController {

    @Autowired
    HelloClient helloClient;

    @RequestMapping(value = "/feign/hello", method = RequestMethod.GET)
    public String feignHello() {
        // hello方法会超时
        String helloResult = helloClient.hello();
        // 获取断路器
        HystrixCircuitBreaker breaker = HystrixCircuitBreaker.Factory
                .getInstance(HystrixCommandKey.Factory
                        .asKey("HelloClient#hello()"));
        System.out.println("断路器状态: " + breaker.isOpen());
        return helloResult;
    }
}
```

在控制器的方法中，获取了hello方法的断路器，并输出其状态。接下来，编写一个测试客户端，多线程访问http://localhost:9000/feign/hello/{index}，也就是控制器的feignHello

方法，客户端请见代码清单 6-27。

代码清单 6-27：06\6.4\spring-hystrix-invoker\src\main\java\org\crazyit\cloud\feign\TestFeignClient.java

```
public class TestFeignClient {

    public static void main(String[] args) throws Exception {
        // 创建默认的 HttpClient
        final CloseableHttpClient httpclient = HttpClients.createDefault();
        // 调用多次服务并输出结果
        for(int i = 0; i < 6; i++) {
            // 建立线程访问接口
            Thread t = new Thread() {
                public void run() {
                    try {
                        String url = "http://localhost:9000/feign/hello";
                        // 调用 GET 方法请求服务
                        HttpGet httpget = new HttpGet(url);
                        // 获取响应
                        HttpResponse response = httpclient.execute(httpget);
                        // 根据响应解析出字符串
                        System.out.println(EntityUtils.toString(response.getEntity()));
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            };
            t.start();
        }
        // 等待完成
        Thread.sleep(15000);
    }
}
```

完成后，依次启动 Eureka 服务器、服务提供者、服务调用者，运行代码清单 6-27，可看到“服务调用者”的控制台输出如下：

```
断路器状态：false
断路器状态：false
断路器状态：false
断路器状态：false
```

```
断路器状态：true
断路器状态：true
```

根据输出可知，断路器已经被打开。

6.4.7 Hystrix 监控

为服务调用者加入 Actuator，可以对服务调用者的健康情况进行实时监控。例如可以看到某个方法的断路器是否打开、当前负载等情况，为服务调用者（spring-hystrix-invoker）加入以下依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
    <version>1.5.3.RELEASE</version>
</dependency>
```

重新启动 spring-hystrix-invoker（9000 端口），访问 http://localhost:9000/hystrix.stream，可以看到 Hystrix 输出的 stream 数据。接下来，新建一个监控的 Web 项目，名称为 hystrix-dashboard，对应的代码路径为 codes\06\6.4\hystrix-dashboard，为该项目加入以下依赖：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-hystrix</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
        <version>1.5.3.RELEASE</version>
    </dependency>
</dependencies>
```

该项目的启动类代码如下：

```
@SpringBootApplication
@EnableHystrixDashboard
public class MyApplication {

    public static void main(String[] args) {
        // 设置启动的服务器端口
        new SpringApplicationBuilder(MyApplication.class).properties(
                "server.port=8082").run(args);
    }
}
```

使用了@EnableHystrixDashboard注解开启Hystrix控制台，启动的端口为8082。完成后，启动整个集群，最后再启动监控项目，在浏览器输入以下地址http://localhost:8082/hystrix，可以看到界面如图6-5所示。

图6-5 Hystrix控制台

在文本框中输入需要监控的地址，本例需要监控的地址是http://localhost:9000/hystrix.stream。单击监控按钮后，可看到界面如图6-6所示。

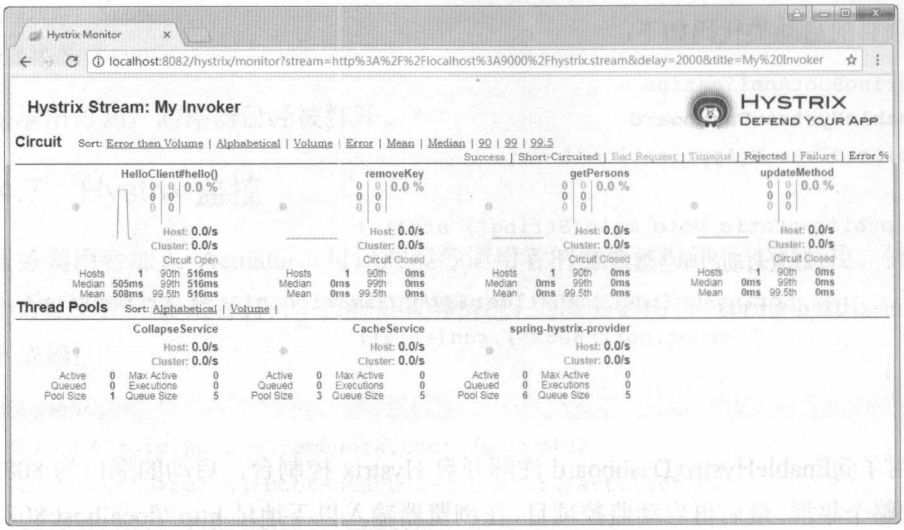


图 6-6 监控界面

如图 6-6 所示，名称为 HelloClient#hello()的命令，断路器被打开，该命令主要用于测试 Feign 的断路器（请见 6.4.6 节）。其他断路器状态正常，读者可以访问本节案例中的各个 URL 来测试这些命令的健康情况。

本节案例所监测的是单节点的健康情况，如果需要监控整个集群的情况，可以使用 Turbine 框架，该框架将在后面章节中讲述。

6.5 本章小结

本章主要讲解了 Hystrix 框架，包括 Hystrix 的使用、运行机制等内容。如果你希望改造旧系统，让其拥有更好的容错能力，可以学习本章的 6.3 节。如果你正在搭建新的项目，想让项目拥有更好的性能，可结合 Spring Cloud 一起使用 Hystrix。

由于 Spring Cloud 对 Hystrix 的封装，使得我们平时在使用 Spring Cloud 时，几乎感觉不到 Hystrix 的存在，但在高并发的场景下，Hystrix 的作用就会很明显。如果能理解 Hystrix 的工作原理，将会对解决实际环境中的复杂问题有很好的帮助。

CHAPTER

7

第 7 章
微服务集群网关

本章要点

- 关于 Zuul
- Zuul 框架的使用
- 在 Spring Cloud 中使用 Zuul
- Zuul 的原理
- Zuul 功能进阶

在前面章节介绍的例子中，我们都是直接访问服务调用者的URL来访问微服务，在实际环境中，应用程序会有多个服务调用者，如何将它们组织起来，统一对外提供服务呢？本章将讲述使用Netflix的Zuul框架构建微服务集群的网关。

7.1 Zuul框架介绍

7.1.1 关于Zuul

Spring Cloud集群提供了多个组件，用于进行集群内部的通信，例如服务管理组件Eureka，负载均衡组件Ribbon。如果集群提供了API或者Web服务，需要与外部进行通信，比较好的方式是添加一个网关，将集群的服务都隐藏到网关后面。这种做法对于外部客户端来说，无须关心集群的内部结构，只需关心网关的配置等信息；对于Spring Cloud集群来说，不必过多暴露服务，提升了集群的安全性。

代理层作为应用集群的大门，在技术选取上尤为重要，很多传统的解决方案，在软件上选择了Nginx、Apache等服务器。Netflix提供了自己的解决方案：Zuul。Zuul是Netflix的一个子项目，Spring Cloud将Zuul进行了进一步的实现与封装，将其整合到spring-netflix项目中，为微服务集群提供代理、过滤、路由等功能。

7.1.2 Zuul的功能

Zuul将外部的请求过程划分为不同的阶段，每个阶段都提供了一系列过滤器，这些过滤器可以帮助我们实现以下功能。

- **身份验证和安全性：**对需要身份认证的资源进行过滤，拒绝处理不符合身份认证的请求。
- **观察和监控：**跟踪重要的数据，为我们展示准确的请求状况。
- **动态路由：**将请求动态路由到不同的服务集群。
- **负载分配：**设置每种请求的处理能力，删除那些超出限制的请求。
- **静态响应处理：**提供一些静态的过滤器，直接响应一些请求，而不将它们转发到集群内部。
- **路由的多样化：**除了可以将请求路由到Spring Cloud集群外，还可以将请求路由到其他服务。

7.2 在 Web 项目中使用 Zuul

本节将带领大家开发一个简单的 Zuul 程序，初步展示 Zuul 的路由功能。

7.2.1 Web 项目整合 Zuul

新建一个名称为 first-router 的 Maven 项目，项目使用的依赖如下：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.5.3</version>
</dependency>
```

需要加入 spring-cloud-starter-zuul 依赖，由于 Zuul 底层使用了 HttpClient，因此还要加入相应的依赖。为了能让 Web 项目开启对 Zuul 的支持，在应用类中加入@EnableZuulProxy 注解，请见代码清单 7-1。

代码清单 7-1：codes\07\02\first-router\src\main\java\org\crazyit\cloud\GatewayApplication.java

```
@EnableZuulProxy
@SpringBootApplication
public class GatewayApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(GatewayApplication.class).properties(
                "server.port=8080").run(args);
    }
}
```

注意该项目的启动端口为 8080。完成以上工作后，一个拥有 Zuul 功能的 Web 项目就建立好了，接下来，将测试它的路由功能。

7.2.2 测试路由功能

前一小节已经建立了路由项目，接下来建立源服务的项目，测试示例的结构请见图 7-1 所示。

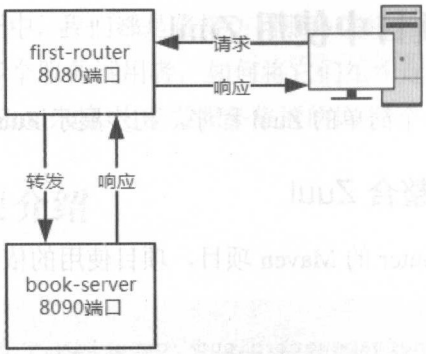


图 7-1 测试示例结构图

新建名称为 book-server 的 Maven 项目，该项目是一个普通的 Spring Boot 项目，使用以下依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>1.5.4.RELEASE</version>
</dependency>
```

为 book-server 添加一个/hello 服务，项目的启动类以及控制器请见代码清单 7-2。

代码清单 7-2：codes\07\02\book-server\src\main\java\org\crazyit\cloud\BookApplication.java

```
@SpringBootApplication
@RestController
public class BookApplication {

    @RequestMapping(value = "/hello/{name}", method = RequestMethod.GET)
    public String hello(@PathVariable String name) {
        return "hello " + name;
    }

    public static void main(String[] args) {
        new SpringApplicationBuilder(BookApplication.class).properties(
                "server.port=8090").run(args);
    }
}
```

为了简单起见，本例将启动类与控制器写到了一起，注意 book-server 的端口为 8090。在控制器中，建立了一个/hello/{name}服务，成功调用后，会返回相应的字符串。接下来，

修改 first-router 项目的配置文件，让其进行转发工作。

修改 first-router 项目的 application.yml 文件，加入以下内容：

```
zuul:
  routes:
    books:
      url: http://localhost:8090
```

加入以上配置后，发送给 http://localhost:8080/books 的所有请求会被转发到 8090 端口，也就是访问 first-router 项目，实际上最终会调用 book-server 的服务。启动两个应用，在浏览器中输入地址 http://localhost:8080/books/hello/crazyit，可以看到浏览器输出如下：

```
hello crazyit
```

根据输出结果可知，发送的请求已经被转发到 book-server 进行处理。

7.2.3 过滤器运行机制

在前面的路由项目中，我们使用了@EnableZuulProxy 注解。开启该注解后，在 Spring 容器初始化时，会将 Zuul 的相关配置初始化，其中包含一个 Spring Boot 的 Bean：ServletRegistrationBean，该类主要用于注册 Servlet。Zuul 提供了一个 ZuulServlet 类，在 Servlet 的 service 方法中，执行各种 Zuul 过滤器（ZuulFilter）。图 7-2 所示为 HTTP 请求在 ZuulServlet 中的生命周期。

ZuulServlet 的 service 方法接收到请求后，会执行 pre 阶段的过滤器，再执行 routing 阶段的过滤器，最后执行 post 阶段的过滤器。其中 routing 阶段的过滤器会将请求转发到“源服务”，源服务可以是第三方的 Web 服务，也可以是 Spring Cloud 的集群服务。在执行 pre 和 routing 阶段的过滤器时，如果出现异常，则会执行 error 过滤器。整个过程的 HTTP 请求、HTTP 响应、状态等数据，都会被封装到一个 RequestContext 对象中，这将在后面章节中讲述。

大致了解了 Zuul 的运行机制后，下面开始讲解如何在 Spring Cloud 中使用 Zuul。

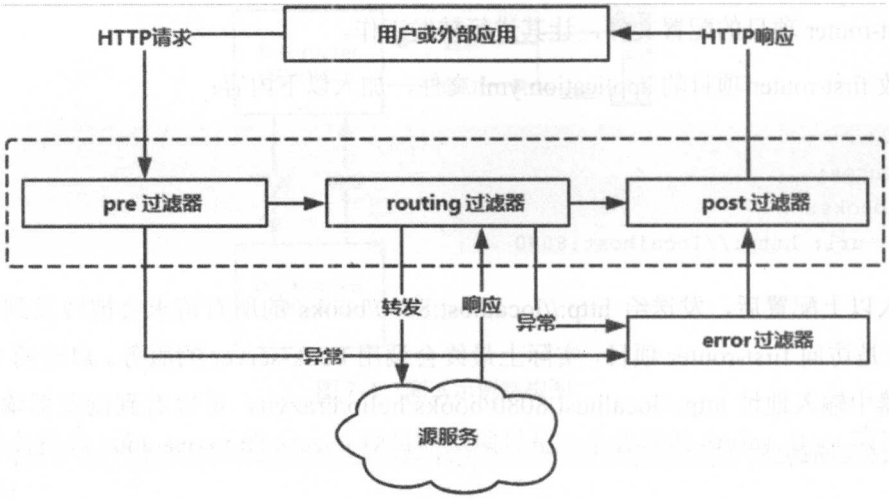


图 7-2 HTTP 请求的生命周期

7.3 在微服务集群中初试 Zuul

在前面小节介绍的例子中，Zuul 将请求转发到一个 Web 项目进行处理，如果实际处理请求的不是一个 Web 项目，而是整个微服务集群，那么 Zuul 将成为整个集群的网关。在加入 Zuul 前，Spring Cloud 集群的结构请见图 7-3。

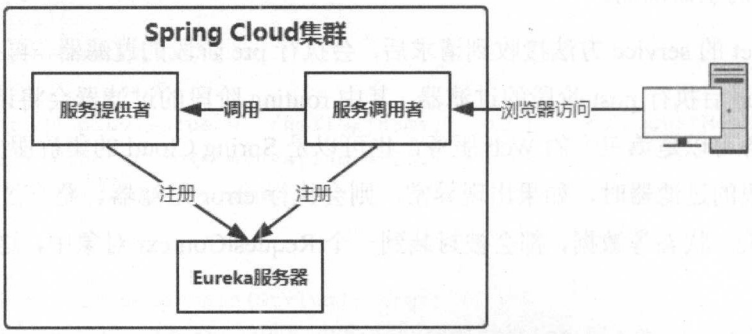


图 7-3 原来的 Spring Cloud 集群结构

为微服务集群加入 Zuul 网关后，结构如图 7-4 所示。

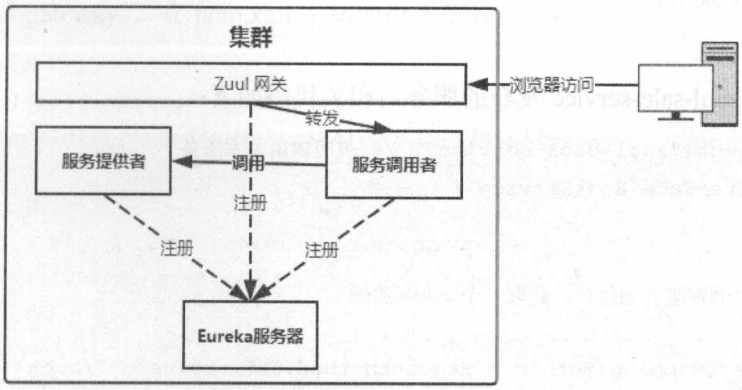


图 7-4 加入 Zuul 后的集群结构

在深入学习 Zuul 前，先按图 7-4 所示搭建本章的测试项目。

7.3.1 集群搭建

假设当前需要实现一个书本销售业务，在销售模块中需要调用书本模块的服务来查找书本，本小节的案例以此为基础，建立以下项目。

- zuul-eureka-server：Eureka 服务器，应用端口为 8761，源代码目录为 codes\07\03\zuul-eureka-server。
- zuul-book-service：书本模块，属于服务提供者，提供/book/{bookId}服务，用于查找图书，最后返回 Book 的 JSON 字符串，应用端口为 9000，代码目录为 codes\07\03\zuul-book-service。
- zuul-sale-service：销售模块，属于服务调用者，对外发布销售服务/sale-book/ {bookId}，在该服务中会调用 zuul-book-service 来查找 Book，应用端口为 9100，代码目录为 codes\07\03\zuul-sale-service。

书本模块 zuul-book-service 发布的服务仅返回一个简单的 Book 对象，控制器代码如下：

```
@RequestMapping(value = "/book/{bookId}", method = RequestMethod.GET,
      produces = MediaType.APPLICATION_JSON_VALUE)
public Book findBook(@PathVariable Integer bookId) {
   Book book = new Book();
   book.setId(bookId);
   book.setName("Workflow 讲义");
   book.setAuthor("杨恩雄");
```

```
    return book;
}
```

销售模块 zuul-sale-service 发布的服务，相关代码如下：

```
@FeignClient("zuul-book-service") // 声明调用书本服务
public interface BookService {

    /**
     * 调用书本服务的接口，获取一个 Book 实例
     */
    @RequestMapping(method = RequestMethod.GET, value = "/book/{bookId}")
    Book getBook(@PathVariable("bookId") Integer bookId);
}

@RestController
public class SaleController {

    @Autowired
    private BookService bookService;

    @RequestMapping(value = "/sale-book/{bookId}", method = RequestMethod.GET)
    public String saleBook(@PathVariable Integer bookId) {
        // 调用 book 服务查找
        Book book = bookService.getBook(bookId);
        // 控制台输入，模拟进行销售
        System.out.println("销售模块处理销售，要销售的图书 id: " + book.getId() + ", 书名: "
                + book.getName());
        // 销售成功
        return "SUCCESS";
    }
}
```

销售模块的服务使用 Feign 调用书本模块的服务来获取 Book 实例，然后在控制台中输出信息。在实际应用中，销售的过程会更为复杂，例如有可能涉及支付等内容，本例为了简单起见，仅进行简单的输出。接下来，创建网关项目。

7.3.2 路由到集群服务

在前一小节的基础上，新建一个名称为 zuul-gateway 的 Maven 项目（代码目录为

codes\07\03\zuul-gateway)，在 pom.xml 中加入以下依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.5.3</version>
</dependency>
```

新建应用类，如代码清单 7-3 所示。

代码清单 7-3：codes\07\03\zuul-gateway\src\main\java\org\crazyit\cloud\GatewayApplication.java

```
@EnableZuulProxy
@SpringBootApplication
public class GatewayApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(GatewayApplication.class).properties(
                "server.port=8080").run(args);
    }
}
```

应用类跟前面的例子一致，使用@EnableZuulProxy 注解。但是，由于网关项目需要加到集群中，因此要修改配置文件，让其注册到 Eureka 服务器中。本例的配置文件如代码清单 7-4 所示。

代码清单 7-4：codes\07\03\zuul-gateway\src\main\resources\application.yml

```
spring:
  application:
    name: zuul-gateway
```

```
eureka:
  instance:
    hostname: localhost
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
zuul:
  routes:
    sale:
      path: /sale/**
      serviceId: zuul-sale-service
```

使用eureka的配置，将自己注册到8761的Eureka中。在配置Zuul时，声明所有的/sale/**请求将会被转发到Id为zuul-sale-service的服务进行处理。

一般情况下，配置了serviceId后，在处理请求的routing阶段，将会使用一个名称为RibbonRoutingFilter的过滤器，该过滤器会调用Ribbon的API来实现负载均衡，默认情况下用HttpClient来调用集群服务。

按照以下顺序启动集群：

- 启动zuul-eureka-server（Eureka服务器）。
- 启动zuul-book-service（服务提供者）。
- 启动zuul-sale-service（服务调用者）。
- 启动zuul-gateway（集群网关）。

在浏览器中访问http://localhost:8080/sale/sale-book/1，返回SUCCESS字符串，在销售模块的控制台，可以看到输出如下：

```
销售模块处理销售，要销售的图书id: 1，书名：Workflow讲义
```

根据输出可知，销售模块、书本模块均被调用。本例涉及4个项目，图7-5展示了本例的结构，可帮助读者理解本例。

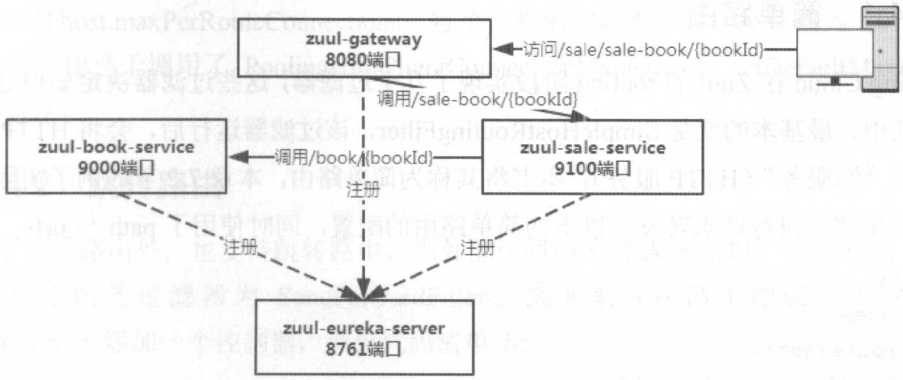


图 7-5 本例的结构

7.3.3 Zuul Http 客户端

我们知道，Ribbon 用来实现负载均衡，Ribbon 在选取了合适的服务器后，再调用 REST 客户端 API 来调用集群服务。在默认情况下，将使用 HttpClient 的 API 来调用集群服务。除了 HttpClient 外，还可以使用 OkHttpClient，以及 com.netflix.niws.client.http.RestClient，RestClient 目前已经不推荐使用，如果想启用 OkHttpClient，可以添加以下配置：ribbon.okhttp.enabled=true。除了该配置外，还要在 pom.xml 中加入 OkHttpClient 的依赖：

```
<dependency>
    <groupId>com.squareup.okhttp3</groupId>
    <artifactId>okhttp</artifactId>
</dependency>
```

注意：

本节中的案例也会在本章后面章节中使用。

7.4 路由配置

路由配置看似简单，但也有部分规则需要说明，本节以 7.3 节搭建的集群项目为基础讲解 Zuul 的路由配置。

7.4.1 简单路由

Spring Cloud 在 Zuul 的 routing 阶段实现了几个过滤器，这些过滤器决定如何进行路由工作。其中，最基本的就是 SimpleHostRoutingFilter，该过滤器运行后，会将 HTTP 请求全部转发到“源服务”（HTTP 服务），本书将其称为简单路由，本章 7.2 节的例子实际上就是使用了简单路由进行请求转发。以下为简单路由的配置，同时使用了 path 与 url：

```
zuul:
  routes:
    routeTest:
      path: /routeTest/163
      url: http://www.163.com
```

以上的配置访问 http://localhost:8080/reuteTest/163，将会跳转到 163 网站。为了配置简便，可以省略 path，默认情况下使用 routeId 作为 path，以下的配置省略了 path 配置：

```
zuul:
  routes:
    route163:
      url: http://www.163.com
```

访问 http://localhost:8080/route163，同样会路由到 163 网站。实际上，要触发简单路由，配置的 url 的值需要以 http:或者 https:字符串开头。以下的配置不能触发简单路由：

```
zuul:
  routes:
    noRoute163:
      url: www.163.com
```

简单路由的过滤器 SimpleHostRoutingFilter 使用 HttpClient 进行转发，该过滤器会将 HttpServletRequest 的相关数据（HTTP 方法、参数、请求头等）转换为 HttpClient 的请求实例（HttpRequest），再使用 CloseableHttpClient 进行转发。

在此过程中，为了保证转发的性能，使用了 HttpClient 的连接池功能。涉及连接池，就需要对其进行配置。在使用简单路由时，可以配置以下两项，修改 HttpClient 连接池的属性。

- zuul.host.maxTotalConnections：目标主机的最大连接数，默认值为 200。配置该项，相当于调用了 PoolingHttpClientConnectionManager 的 setMaxTotal 方法。

- zuul.host.maxPerRouteConnections：每个主机的初始连接数，默认值为 20。配置该项，相当于调用了 PoolingHttpClientConnectionManager 的 setDefaultMaxPerRoute 方法。

7.4.2 跳转路由

除了简单路由外，也支持跳转路由。当外部访问网关的 A 地址时，会跳转到 B 地址，处理跳转路由的过滤器为 SendForwardFilter。接下来进行简单测试，为网关项目（zuul-gateway）添加一个控制器，请见代码清单 7-5。

代码清单 7-5：codes\07\03\zuul-gateway\src\main\java\org\crazyit\cloud\web\SourceController.java

```
@RestController
public class SourceController {

    @RequestMapping(value = "/source/hello/{name}", method = RequestMethod.GET)
    public String hello(@PathVariable("name") String name) {
        return "hello " + name;
    }
}
```

控制器中提供了一个最简单的 hello 服务，用来当作“源服务”，在 application.yml 中进行转发配置，配置项如下：

```
zuul:
  routes:
    helloRoute:
      path: /test/**
      url: forward:/source/hello
```

当外部访问/test 地址时，将会自动跳转到/source/hello 地址。打开浏览器，输入 http://localhost:8080/test/anugs，可以看到浏览器会返回字符串 hello angus，可见源服务被调用。跳转路由实现较为简单，实际上是调用了 RequestDispatcher 的 forward 方法进行跳转。

7.4.3 Ribbon 路由

在 7.3.2 节中，我们已经接触过 Ribbon 路由。当网关作为 Eureka 客户端注册到 Eureka 服务器时，可以通过配置 serviceId 将请求转发到集群的服务中。使用以下配置，可以执行 Ribbon 路由过滤器：

```
zuul:
  routes:
    sale:
      path: /sale/**
      serviceId: zuul-sale-service
```

与简单路由类似，serviceId 也可以被省略。当省略时，将会使用 routeId 作为 serviceId，下面的配置片断，效果等同于上面的配置：

```
zuul:
  routes:
    zuul-sale-service:
      path: /sale/**
```

需要注意的是，如果提供的 url 配置项不是简单路由格式（不以 http:或 https:开头），也不是跳转路由格式（forward:开头），那么将会执行 Ribbon 路由过滤器，将 url 看作一个 serviceId。下面的配置片断，效果也等同于前面的配置：

```
zuul:
  routes:
    sale:
      path: /sale/**
      url: zuul-sale-service
```

7.4.4 自定义路由规则

如果上面的路由配置无法满足实际需求，可以考虑使用自定义的路由规则。实现方式较为简单，在配置类中创建一个 PatternServiceRouteMapper 即可，请见代码清单 7-6。

代码清单 7-6：codes\07\03\zuul-gateway\src\main\java\org\crazyit\cloud\MyConfig.java

```
@Configuration
public class MyConfig {

    @Bean
    public PatternServiceRouteMapper patternServiceRouteMapper() {
        return new PatternServiceRouteMapper(
                "(zuul)-(?<module>.+)-(service)", "${module}/**");
    }
}
```

创建了 PatternServiceRouteMapper 实例，构造器的第一个参数为 serviceId 的正则表达

式，第二个参数为路由的 path。访问 module/**的请求，将会被路由到 zuul-module-service 的微服务。

更进一步，以上的路由规则，如果想让一个或多个服务不被路由，可以使用 zuul.ignoredServices 属性。例如在代码清单 7-6 的基础上，想排除 zuul-sale-service、zuul-book-service 这两个模块，可以配置 zuul.ignoredServices: zuul-sale-service, zuul-book-service。

7.4.5 忽略路由

除了上面提到的 zuul.ignoredServices 配置可以忽略路由外，还可以使用 zuul.ignoredPatterns 来设置不进行路由的 URL，请见以下配置片断：

```
zuul:
  ignoredPatterns: /sale/noRoute
  routes:
    sale:
      path: /sale/**
      serviceId: zuul-sale-service
```

访问/sale 路径的请求都会被路由到 zuul-sale-service 进行处理，但/sale/noRoute 除外。

7.5 Zuul 的其他配置

本节将讲解 Zuul 一些较为常用的配置。

7.5.1 请求头配置

在集群的服务间共享请求头并没有什么问题，但是如果请求会被转发到其他系统，那么对于敏感的请求头信息，就需要进行处理。在默认情况下，HTTP 请求头的 Cookie、Set-Cookie、Authorization 属性不会传递到“源服务”，可以使用 sensitiveHeaders 属性来配置敏感请求头，下面的配置对全局生效：

```
zuul:
  sensitiveHeaders: accept-language, cookie
```

以下的配置片断，仅对一个路由生效：

```
zuul:
```

```
  routes:
    sale:
      path: /sale/**
      serviceId: zuul-sale-service
      sensitiveHeaders: cookie
```

除了使用 sensitiveHeaders 属性外，还可以使用 ignoredHeaders 属性来配置全局忽略的请求头。使用该配置项后，请求与响应中所配置的头信息均被忽略：

```
zuul:
  ignoredHeaders: accept-language
```

7.5.2 路由端点

在网关项目中提供了一个/routes 服务，可以让我们查看路由映射信息。如果想开启该服务需要满足以下条件：

- 网关项目中引入了 Spring Boot Actuator。
- 项目中使用了@EnableZuulProxy 注解。

一般情况下，Actuator 开启了端点的安全认证，即使符合以上两个条件，也无法访问 routes 服务。要解决该问题，可以在配置文件中将 management.security.enabled 属性值设置为 false 关闭安全认证。

以 7.3 节中介绍的项目为例，开启/routes 服务后访问 http://localhost:8080/routes，浏览器中将输出以下 JSON（以下 JSON 经过格式化）：

```
{
    "/sale/**": "zuul-sale-service",
    "/routeTest/163": "http://www.163.com",
    "/route163/**": "http://www.163.com",
    "/noRoute163/**": "www.163.com",
    "/test/**": "forward:/source/hello",
    "/zuul-sale-service/**": "zuul-sale-service",
    "/zuul-book-service/**": "zuul-book-service"
}
```

7.5.3 Zuul 与 Hystrix

当我们对网关进行配置让其调用集群的服务时，将会执行 Ribbon 路由过滤器（RibbonRoutingFilter）。该过滤器在进行转发时会封装为一个 Hystrix 命令予以执行。换言

之，它具有容错的功能。如果“源服务”出现问题（例如超时），那么所执行的 Hystrix 命令将会触发回退，下面将测试 Zuul 中的回退。

为销售模块（zuul-sale-service）的控制器添加一个超时方法，请见代码清单 7-7。

代码清单 7-7：codes\07\03\zuul-sale-service\src\main\java\org\crazyit\cloud\web\SaleController.java

```
@RequestMapping(value = "/errorTest", method = RequestMethod.GET)
public String errorTest() throws Exception {
    Thread.sleep(3000);
    return "errorTest";
}
```

在网关项目（zuul-gateway）中建立一个网关处理类，处理回退逻辑，请见代码清单 7-8。

代码清单 7-8：codes\07\03\zuul-gateway\src\main\java\org\crazyit\cloud\hy\MyFallbackProvider.java

```
public class MyFallbackProvider implements ZuulFallbackProvider {

    // 返回路由的名称
    public String getRoute() {
        return "zuul-sale-service";
    }

    // 回退触发时，返回默认的响应
    public ClientHttpResponse fallbackResponse() {
        return new ClientHttpResponse() {

            public InputStream getBody() throws IOException {
                return new ByteArrayInputStream("fallback".getBytes());
            }

            public HttpHeaders getHeaders() {
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.TEXT_PLAIN);
                return headers;
            }

            public HttpStatus getStatusCode() throws IOException {
                return HttpStatus.OK;
            }

            public int getRawStatusCode() throws IOException {
```

```
                return 200;
            }

            public String getStatusText() throws IOException {
                return "OK";
            }

            public void close() {

            }
        };
    }
}
```

回退处理类需要实现 ZuulFallbackProvider 接口，实现的 getRoute 方法返回路由的名称，该方法将与配置中的路由进行对应，本例配置的路由如下：

```
zuul:
  routes:
    sale:
      path: /sale/**
      serviceId: zuul-sale-service
```

简单点说就是，zuul-sale-service 路由出现问题导致触发回退时，由 MyFallbackProvider 处理。MyFallbackProvider 类实现的 fallbackResponse 方法要返回一个 ClientHttpResponse 实例。本例中返回的 ClientHttpResponse，内容为 fallback，也就是回退触发时，调用的客户端将得到 fallback 字符串。

为了让 Spring 容器知道 MyFallbackProvider，在配置类中新建 MyFallbackProvider 的 Bean，如代码清单 7-9 所示。

代码清单 7-9：codes\07\03\zuul-gateway\src\main\java\org\crazyit\cloud\hy\FallbackConfig.java

```
@Configuration
public class FallbackConfig {

    @Bean
    public ZuulFallbackProvider saleFallbackProvider() {
        return new MyFallbackProvider();
    }
}
```

启动整个集群，在浏览器中访问以下地址 http://localhost:8080/sale/errorTest，浏览器返回 fallback 字符串，可见回退被触发。

以上实现的 MyFallbackProvider 仅对 zuul-sale-service 路由有效，如果想对全局有效，可以使用以下实现：

```
public String getRoute() {
    return "*";
}
```

7.5.4 在 Zuul 中预加载 Ribbon

调用集群服务时，会使用 Ribbon 的客户端。默认情况下，客户端相关的 Bean 会延迟加载，在第一次调用集群服务时，才会初始化这些对象。在第一次调用时，控制台会有以下的输出日志（仅截取部分）：

```
2017-08-28 18:44:31.963  INFO 528 --- [          main]
c.n.l.DynamicServerListLoadBalancer     :
DynamicServerListLoadBalancer for client zuul-sale-service initialized:
```

如果想提前加载 Ribbon 客户端，可以在配置文件中进行以下配置：

```
zuul:
  ribbon:
    eager-load:
      enabled: true
```

以上的配置在 Spring 容器初始化时，就会创建 Ribbon 客户端的相关实例。启动网关项目可以看到以上的输出日志。

至此，Zuul 的基本功能已经介绍完毕。掌握了前面章节介绍的内容，基本上就可以使用 Zuul 了。接下来，再进一层，让我们更深入地学习 Zuul。

7.6 Zuul 功能进阶

本节的内容涉及 Zuul 的原理，如果读者只求掌握 Zuul 的使用，可以跳过本节。

7.6.1 过滤器优先级

Spring Cloud 为 HTTP 请求的各个阶段提供了多个过滤器，这些过滤器的执行顺序由它

们各自提供的一个 int 值决定，提供的值越小，优先级越高。图 7-6 展示了默认的过滤器，以及它们的优先级。

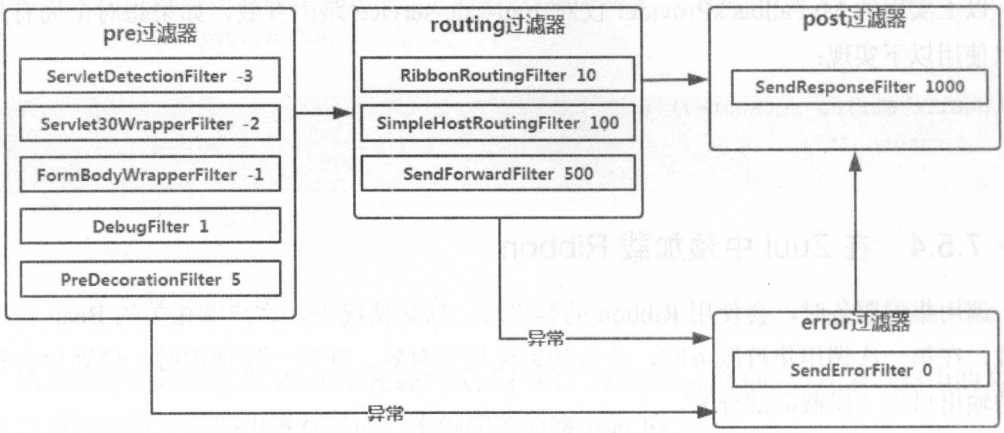


图 7-6　Spring Cloud 自带的过滤器及优先级

如图 7-6 所示，在 routing 阶段会优先执行 Ribbon 路由的过滤器，再执行简单路由过滤器。

7.6.2　自定义过滤器

了解过滤器的执行顺序后，我们编写一个自定义过滤器。新建过滤类，继承 ZuulFilter，实现请见代码清单 7-10。

代码清单 7-10：codes\07\03\zuul-gateway\src\main\java\org\crazyit\cloud\filter\MyFilter.java

```
public class MyFilter extends ZuulFilter {

    // 过滤器执行条件
    public boolean shouldFilter() {
        return true;
    }

    // 执行方法
    public Object run() {
        System.out.println("执行 MyFilter 过滤器");
        return null;
    }
```

```
    // 表示将在路由阶段执行
    public String filterType() {
        return FilterConstants.ROUTE_TYPE;
    }

    // 返回1，路由阶段，该过滤将会最先执行
    public int filterOrder() {
        return 1;
    }
}
```

新建的自定义过滤器将会在 routing 阶段执行，优先级为 1，也就是在 routing 阶段，该过滤器最先执行。另外注意 shouldFilter 方法，过滤最终是否执行由该方法决定，本例返回 true，表示访问任何路由规则都会执行该过滤器。为了让 Spring 容器知道过滤器的存在，需要对该类进行配置，代码清单 7-11 所示为配置类。

代码清单 7-11：codes\07\03\zuul-gateway\src\main\java\org\crazyit\cloud\filter\FilterConfig.java

```
@Configuration
public class FilterConfig {

    @Bean
    public MyFilter myFilter() {
        return new MyFilter();
    }
}
```

启动集群，访问网关 http://localhost:8080/test/1，会看到控制输出：执行 MyFilter 过滤器。实际上，访问任何一个配置好的路由都会进行输出。

7.6.3 动态加载过滤器

相对于集群中的其他节点，网关更需要长期、稳定地提供服务。如果需要增加过滤器，重启网关代价太大，为了解决该问题，Zuul 提供了过滤器的动态加载功能。可以使用 Groovy 来编写过滤器，然后添加到加载目录，让 Zuul 去动态加载。先为网关项目加入 Groovy 的依赖：

```
<dependency>
    <groupId>org.codehaus.groovy</groupId>
    <artifactId>groovy-all</artifactId>
    <version>2.4.12</version>
</dependency>
```

接下来，在网关项目的应用类中，调用 Zuul 的 API 来实现动态加载，请见代码清单 7-12。

代码清单 7-12：codes\07\03\zuul-gateway\src\main\java\org\crazyit\cloud\GatewayApplication.java

```
@EnableZuulProxy
@SpringBootApplication
public class GatewayApplication {

    @PostConstruct
    public void zuulInit() {
        FilterLoader.getInstance().setCompiler(new GroovyCompiler());
        // 读取配置，获取脚本根目录
        String scriptRoot = System.getProperty("zuul.filter.root", "groovy/ filters");
        // 获取刷新间隔
        String refreshInterval = System.getProperty("zuul.filter.refreshInterval", "5");
        if (scriptRoot.length() > 0) scriptRoot = scriptRoot + File.separator;
        try {
            FilterFileManager.setFilenameFilter(new GroovyFileFilter());
            FilterFileManager.init(Integer.parseInt(refreshInterval), scriptRoot+"pre",
                    scriptRoot + "route", scriptRoot + "post");
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

    }

    public static void main(String[] args) {
        new SpringApplicationBuilder(GatewayApplication.class).properties(
                "server.port=8080").run(args);
    }
}
```

在启动类中，增加了 zuulInit 方法，使用@PostConstruct 进行修饰。在该方法中，先读取 zuul.filter.root 和 zuul.filter.refreshInterval 两个属性，分别表示动态过滤器的根目录以及刷新间隔，刷新间隔以秒为单位。这两个属性优先读取配置文件的值，如果没有则使用默认值。在配置文件中，可使用下面的配置片断：

```
zuul:
  filter:
```

```
    root: "groovy/filters"
    refreshInterval: 5
```

调用 FilterFileManager 的 init 方法，初始化 3 个过滤器目录：pre、route 和 post。为了测试动态加载，使用 Groovy 编写一个最简单的过滤器，请见代码清单 7-13。

代码清单 7-13：codes\07\03\zuul-gateway\src\main\java\groovy\filters\DynamicFilter.groovy

```
class DynamicFilter extends ZuulFilter {

    public boolean shouldFilter() {
        return true;
    }

    public Object run() {
        System.out.println("========= 这一个是动态加载的过滤器：DynamicFilter");
        return null;
    }

    public String filterType() {
        return FilterConstants.ROUTE_TYPE;
    }

    public int filterOrder() {
        return 3;
    }
}
```

与前面的过滤器一致，同样继承自 ZuulFilter。需要注意的是，本例的过滤器并没有一开始就放到动态加载的过滤器目录中，读者在测试时，需要先启动网关项目，再将 DynamicFilter.groovy 放到对应目录中。

完成以上工作后，启动网关项目，访问以下地址 http://localhost:8080/test/crazyit，控制台中并没有输出 DynamicFilter 的信息。将 DynamicFilter.groovy 复制到 src/main/java/groovy/filters/ route 目录，等待几秒后，重新访问以上地址，可以看到网关的控制台输出如下：

```
========= 这一个是动态加载的过滤器：DynamicFilter
```

7.6.4 禁用过滤器

如果想禁用其中一个过滤器，可以使用以下配置：

```
zuul:
  SendForwardFilter:
    route:
      disable: true
```

以上配置会将 SendForwardFilter（处理跳转路由的过滤器）禁用，如果再为 url 属性使用 forward:进行配置的话，将不会产生跳转效果。同样，禁同其他过滤器也会导致失去相应的功能。

7.6.5 请求上下文

HTTP 请求的全部信息都封装在一个 RequestContext 对象中，该对象继承 ConcurrentHashMap。可将 RequestContext 看作一个 Map，RequestContext 维护着当前线程的全部请求变量，例如请求的 URI、serviceId、主机等信息。**本小节将以 RequestContext 为基础，编写一个自定义的过滤器，使用 RestTemplate 来调用集群服务。**

新建一个过滤器，实现请见代码清单 7-14。

代码清单 7-14：codes\07\03\zuul-gateway\src\main\java\org\crazyit\cloud\filter\RestTemplateFilter.java

```
public class RestTemplateFilter extends ZuulFilter {

    private RestTemplate restTemplate;

    public RestTemplateFilter(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    public boolean shouldFilter() {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();
        // 获取请求 uri
        String uri = request.getRequestURI();
        // 为了不影响其他路由，uri 中含有 rest-tpl-sale 才执行本路由器
        if(uri.indexOf("rest-tpl-sale") != -1) {
            return true;
        } else {
            return false;
        }
    }
```

```
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        // 获取需要调用的服务 id
        String serviceId = (String)ctx.get("serviceId");
        // 获取请求的 uri
        String uri = (String)ctx.get("requestURI");
        // 组合成 url 给 RestTemplate 调用
        String url = "http://" + serviceId + uri;
        System.out.println("执行 RestTemplateFilter, 调用的 url: " + url);
        // 调用并获取结果
        String result = this.restTemplate.getForObject(url, String.class);
        // 设置路由状态，表示已经进行路由
        ctx.setResponseBody(result);
        // 设置响应标识
        ctx.sendZuulResponse();
        return null;
    }

    @Override
    public String filterType() {
        return FilterConstants.ROUTE_TYPE;
    }

    @Override
    public int filterOrder() {
        return 2;
    }
}
```

RestTemplateFilter 的主要功能是使用 RestTemplate 来调用集群服务。过滤器中的 shouldFilter 方法从 RequestContext 中获取 HttpServletRequest，再得到请求的 uri，如果 uri 含有 rest-tpl-sale 字符串，才执行本过滤器，这样做是为了避免影响其他例子的运行效果。

RestTemplateFilter 实现的 filterType 方法表示该过滤器将在 routing 阶段执行，执行顺序为 2，也就是比 Spring Cloud 自带的过滤器（routing 阶段）都要优先执行。

在 RestTemplateFilter 的执行方法中，从 RequestContext 中获取了 serviceId 以及请求的 uri，再组合成一个 url 给 RestTemplate 执行，执行返回的结果被设置到 RequestContext 中。

需要注意的是，最后调用了 RequestContext 的 sendZuulResponse 方法来设置响应标识。

调用了该方法后，Spring Cloud 自带的 Ribbon 路由过滤器（RibbonRoutingFilter）、简单路由过滤器（SimpleHostRoutingFilter）将不会执行。

将 RestTemplateFilter 加入配置中，请见代码清单 7-15。

代码清单 7-15：codes\07\03\zuul-gateway\src\main\java\org\crazyit\cloud\filter\FilterConfig.java

```
@Bean
public RestTemplateFilter restTemplateFilter(RestTemplate restTemplate) {
    return new RestTemplateFilter(restTemplate); // 注入 RestTemplate
}

@LoadBalanced
@Bean
public RestTemplate getRestTemplate() {
    return new RestTemplate();
}
```

在 application.yml 文件中，建立对应的路由规则，请见以下配置片断：

```
zuul:
  routes:
    restTestRoute:
      path: /rest-tpl-sale/**
      serviceId: zuul-sale-service
```

以上配置片断，设置路由的 path 为/rest-tpl-sale，当访问该地址时，将会执行前面的 RestTemplateFilter。启动集群，访问以下地址：http://localhost:8080/rest-tpl-sale/sale-book/1，浏览器输出返回的字符串 SUCCESS，控制台输出如下：

```
执行 RestTemplateFilter， 调用的 url：http://zuul-sale-service/sale-book/1
```

根据结果可知，我们自定义的过滤器将请求路由到集群的 zuul-sale-service 服务。本例的作用，除了再次展示如何编写过滤器之外，主要还想让大家了解 RequestContext 所维护的相关信息。

7.6.6 @EnableZuulServer 注解

在本章前面的网关项目中，使用了@EnableZuulProxy 来开启 Zuul 的功能。除了该注解外，还可以使用@EnableZuulServer，该注解更像一个“低配版”的@EnableZuulProxy。使用@EnableZuulServer 后，SimpleHostRoutingFilter、RibbonRoutingFilter 等过滤器将不会被

启用。图 7-7 展示了使用@EnableZuulServer 注解后各阶段的过滤器。

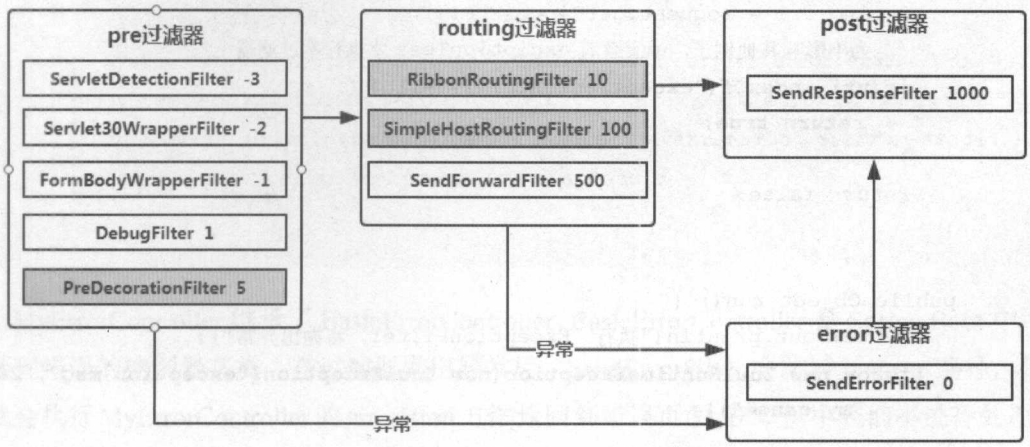


图 7-7 不启用的过滤器

如图 7-7 所示，使用@EnableZuulServer 后，pre 阶段的 PreDecorationFilter，routing 阶段的 RibbonRoutingFilter 和 SimpleHostRoutingFilter 将不会启用。换言之，默认情况下 Zuul 不具备调用集群服务的能力，也不具备简单路由的功能。

如果在实际项目中不希望使用 Spring Cloud 的 RibbonRoutingFilter 和 SimpleHostRoutingFilter，而想像 7.6.4 节那样，自己编写过滤器来调用服务，可以考虑使用@EnableZuulServer 注解。

7.6.7 error 过滤器

各阶段的过滤器执行时，抛出的异常会被捕获，然后调用 RequestContext 的 setThrowable 方法设置异常。error 阶段的 SendErrorFilter 过滤器会判断 RequestContext 中是否存在异常（getThrowable 是否为 null），如果存在，才会执行 SendErrorFilter 过滤器。

SendErrorFilter 过滤器在执行时，会将异常信息设置到 HttpServletRequest 中，再调用 RequestDispatcher 的 forward 方法，默认跳转到/error 页面。代码清单 7-16 编写了一个自定义的过滤器，该过滤器会抛出异常。

代码清单 7-16：codes\07\03\zuul-gateway\src\main\java\org\crazyit\cloud\filter\ExceptionFilter.java

```
public class ExceptionFilter extends ZuulFilter {

    public boolean shouldFilter() {
        RequestContext ctx = RequestContext.getCurrentContext();
```

```
        HttpServletRequest request = ctx.getRequest();
        // 获取请求的uri
        String uri = request.getRequestURI();
        // 为不影响其他例子，uri含有exceptionTest才执行本过滤器
        if(uri.indexOf("exceptionTest") != -1) {
            return true;
        }
        return false;
    }

    public Object run() {
        System.out.println("执行  ExceptionFilter, 将抛出异常");
        throw new ZuulRuntimeException(new ZuulException("exception msg", 201,
            "my cause"));
    }

    public String filterType() {
        return FilterConstants.ROUTE_TYPE;
    }

    public int filterOrder() {
        return 3;
    }
}
```

在ExceptionFilter的shouldFilter方法中，遇到exceptionTest的uri才会执行，目的是不影响本章其他例子的执行。在run方法中，简单进行控制台打印，再抛出一个ZuulRuntimeException，该异常实例包装了一个ZuulException。为了查看异常输出的信息，新建一个控制器，主要在控制台中输出这些信息。请见代码清单7-17。

代码清单7-17：codes\07\03\zuul-gateway\src\main\java\org\crazyit\cloud\web\MyErrorController.java

```
@Controller
public class MyErrorController extends BasicErrorController {

    public MyErrorController(ErrorAttributes errorAttributes) {
        super(errorAttributes, new ErrorProperties());
    }

    @Override
```

```
    public ModelAndView errorHtml(HttpServletRequest request,
            HttpServletResponse response) {
        System.out.println("===  输出异常信息  ===");
        System.out.println(request.getAttribute("javax.servlet.error.status_code"));
        System.out.println(request.getAttribute("javax.servlet.error.exception"));
        System.out.println(request.getAttribute("javax.servlet.error.message"));
        return super.errorHtml(request, response);
    }
}
```

MyErrorController 继承了 BasicErrorController，BasicErrorController 是 Spring Boot 中用于处理错误的控制器基类。在过滤器抛出异常后，SendErrorFilter 会跳转到/error 路径，然后就会执行 MyErrorController 的 errorHtml 方法返回到错误页面。在本例中我们不进行处理，只在方法体中输出此处得到的异常信息。启动整个集群，访问以下地址 http://localhost:8080/exceptionTest/test，可以看到网关项目的控制台输出如下：

```
执行  ExceptionFilter，将抛出异常
2017-08-29 13:25:34.695  WARN 5956 --- [nio-8080-exec-1] o.s.c.n.z.filters.
post.SendErrorFilter   : Error during filtering
……省略中间的异常信息
===  输出异常信息  ===
201
com.netflix.zuul.exception.ZuulException: exception msg
my cause
```

根据输出结果可知，过滤器抛出的异常信息可以在错误处理的控制器中获取。

7.6.8 动态路由

在前面章节中，所有的路由规则都在 application.yml 中进行配置，在实际应用中，可能一个模块就有一份路由配置文件，而且这些配置文件的内容都在不停变化。如果因为部分变化而重启网关，这是无法想象的。因此，路由规则的动态刷新功能在实际应用中非常重要。

路由的动态刷新需要以配置文件的更新、配置项的刷新为前提，这部分内容将在 Spring Cloud Config 章节中讲解，因此动态路由的实现，也在那一章中讲解，本章不进行讲述。

7.7 本章小结

本章以 Zuul 框架为核心，讲解了 Spring Cloud 集群中网关的功能。主要演示了在 Web 项目、在 Spring Cloud 中使用 Zuul，请求转发、微服务调用等内容较为重要，不仅要学会如何使用，最好还要知道其实现原理。本章的 7.4 节与 7.5 节介绍了 Zuul 的常用配置，掌握这些配置后，基本就可以使用 Zuul 了。7.6 节讲解了过滤器的相关内容，学习该节后，可以清楚地了解过滤器的工作机制，以便在实际环境中实现自己所需要的功能。

CHAPTER

8

第8章 微服务与消息驱动

本章要点

- Spring Cloud Stream 介绍
- RabbitMQ 框架
- Apache Kafka 框架
- 开发消息微服务

早在 EJB 2.0 时代，Java EE 引入了 Message Driven Bean（消息驱动 Bean），用于处理企业组件间的消息通信。Spring Cloud 也提供了相关的模块，基于这些模块，可以在微服务中构建消息应用，让微服务可以与其他内部或外部组件通过消息进行通信。本章将以 Spring Cloud Stream 框架为基础，讲解微服务与消息驱动的知识。

8.1 Spring Cloud Stream 介绍

先了解一下 Spring Cloud Stream 框架（下面简称其为 Stream）。

8.1.1 关于 Stream 框架

Spring Cloud Stream 是一个用于构建消息驱动微服务的框架，该框架在 Spring Boot 的基础上整合了 Spring Integration 来连接消息代理中间件。它支持多个消息中间件的自定义配置，同时吸收了这些消息中间件的部分概念，例如持久化订阅、消费者分组和分区等概念。

使用 Stream 框架，我们不必关心如何连接各个消息代理中间件，也不必关心如何进行消息的发送与接收，只需简单地进行配置就可以实现这些功能，让我们可以更加专注于具体的业务逻辑。

8.1.2 Stream 框架的组成部分

Spring Cloud Stream 主要简化了消息应用的开发，该框架主要包括以下内容：

- Stream 框架自己的应用模型。
- 绑定器抽象层，可以与消息代理中间件进行绑定。通过绑定器的 API，可实现插件式的绑定器。
- 持久化订阅的支持。
- 消费者组的支持。
- Topic 分区的支持。

8.1.3 消息代理中间件

目前市面上有许多消息代理中间件（下面简称为消息代理），例如 ActiveMQ、RabbitMQ、Kafka 等。在使用这些框架时，我们需要调用它们的 API，用来发送、接收消息，大概的结

构请见图 8-1。

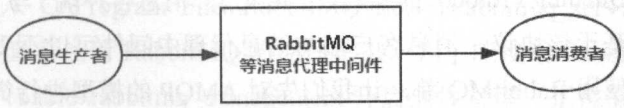


图 8-1 消息代理

消息代理主要用于接收和转发消息，可以把它看作邮局，将消息生产者看作寄件人，将消息消费者看作收件人。我们将一封信件放到邮箱中，邮递员最终会把信件送给收件人，RabbitMQ 等框架进行的就是邮箱、邮递员和邮局的工作，但它们处理的不是信件，而是“消息”。

Spring Cloud Stream 在生产者和消费者之间加入了一个类似代理的角色，它直接与消息代理（邮局）进行交互，消息生产者（寄件人）与消息消费者（收件人）不再需要直接调用各个消息代理框架的 API，它们甚至感觉不到消息代理的存在，类似于图 8-2 所示的结构。

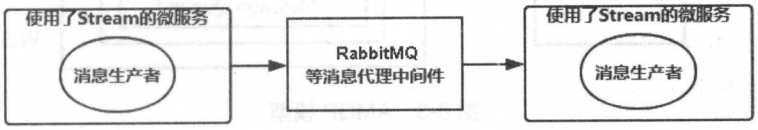


图 8-2 使用了 Spring Cloud Stream 后的结构

使用了 Stream 框架后，消息的生产者和消费者可以更加专注于各自的业务，至于消息（信件）是如何投递的、使用的是哪个消息代理（邮局），它们无须关心。

目前，Spring Cloud Stream 的绑定器提供了 RabbitMQ 与 Kafka 两个消息代理中间件的实现。接下来，我们简单了解一下 RabbitMQ 与 Kafka 框架。

8.2 RabbitMQ 框架

本节将对 RabbitMQ 进行简单介绍，让读者初步了解 RabbitMQ 的作用，读者掌握基本的概念即可。

8.2.1 RabbitMQ 和 AMQP

RabbitMQ 是一个轻量级的消息代理中间件，它支持多种消息通信协议，支持分布式部署，同时也支持运行于多个操作系统，具有灵活、高可用等特性，因此 RabbitMQ 广受欢迎。

RabbitMQ 支持多种协议，其中最为重要的是高级消息队列协议（AMQP），AMQP 是 Advanced Message Queuing Protocol 的缩写，它定义了“消息客户端”与“消息代理中间件”之间的通信协议。基于该协议，消息客户端与消息代理中间件可以不受开发语言、具体产品的约束。在学习使用 RabbitMQ 前，让我们先对 AMQP 的模型进行简单了解。

在前一节中介绍过，消息生产者会向消息代理投递消息，消息代理会将消息再发送给消息消费者，现在对消息代理进行进一步细化，AMQP 的大致模型请见图 8-3。

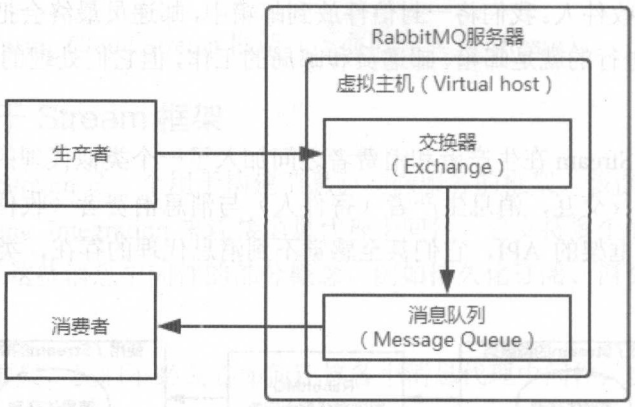


图 8-3 AMQP 模型

在图 8-3 中，生产者会将消息投递给“消息代理（RabbitMQ 服务器）”，它们之间会建立消息通道（Channel），消息由交换器（Exchange）先进行处理，交换器会选择把消息交给哪一个队列（Message Queue），最后消息队列会将消息发给消费者。接下来，先下载和运行 RabbitMQ 服务器。

8.2.2 下载与运行

本章所使用的 RabbitMQ 服务器版本为 3.6.11（Windows），由于 RabbitMQ 服务器使用的是 Erlang 语言，因此还要下载 Erlang，本章所使用的 Erlang 版本为 20.0（64 位）。下载 Erlang 后，会得到 otp_win64_20.0.exe 这样的安装文件，安装文件可以在本书的 soft 目录中找到，笔者已经为大家下载了 32 位和 64 位的安装程序。

注意：

如果在 64 位的 Windows 系统中安装 32 位的 Erlang，在使用 RabbitMQ 时会出现异常。

依次安装完 Erlang 和 RabbitMQ 后，RabbitMQ 会作为 Windows 服务启动。通过 Windows 命令行进入 C:\Program Files\RabbitMQ Server\rabbitmq_server-3.6.11\sbin，输入 rabbitmq-plugins list 查看当前 RabbitMQ 的插件状态。为了能使用 RabbitMQ 的控制台，输入 rabbitmq-plugins enable rabbitmq_management 开启插件管理，正常情况下会输出以下信息：

```
Applying plugin configuration to rabbit@AY-PC... nothing to do.
```

打开浏览器，访问以下地址 http://localhost:15672，可以看到 RabbitMQ 的登录界面，默认的用户名和密码均为 guest，登录后的主界面如图 8-4 所示。

图 8-4 RabbitMQ 主界面

此时，RabbitMQ 的服务器已经成功运行，该服务器就是一个消息代理中间件。接下来，我们可以编写消息的生产者与消费者，使用 RabbitMQ 的 API 进行消息的测试。

8.2.3 编写生产者

相对于服务器来说，消息生产者与消费者都属于客户端，它们与服务器之间通过 AMQP 协议进行通信。AMQP 不受语言的限制，换言之，客户端可以使用不同的编程语言实现，本例将使用 Java 来编写客户端。新建名称为 rabbit-test 的 Maven 项目，加入以下依赖：

```
<dependency>
```

```
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>4.2.0</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.9</version>
</dependency>
```

新建消息生产者，用于向服务器发送消息，请见代码清单 8-1。

代码清单 8-1：codes\08\8.2\rabbit-test\src\main\java\org\crazyit\jms\Send.java

```
public class Send {

    public static void main(String[] args) throws Exception {
        // 创建连接
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        Connection connection = factory.newConnection();
        // 建立通道
        Channel channel = connection.createChannel();
        // 声明队列
        String queueName = "hello";
        channel.queueDeclare(queueName, false, false, false, null);
        String message = "Hello World!";
        // 进行消息发布
        channel.basicPublish("", queueName, null, message.getBytes());
        // 关闭通道和连接
        channel.close();
        connection.close();
    }
}
```

在生产者中，先创建服务器连接，再建立通道和队列，最后向服务器发布一条“Hello World”的消息。细心的读者可能会发现，与 AMQP 的模型图相比，以上代码缺少了交换器（Exchange）。实际上，如果没有声明交换器，就会使用默认的交换器。运行代码

清单 8-1，再打开 RabbitMQ 的主界面，单击 Queues 选项卡，可以看到创建了 hello 队列，如图 8-5 所示。

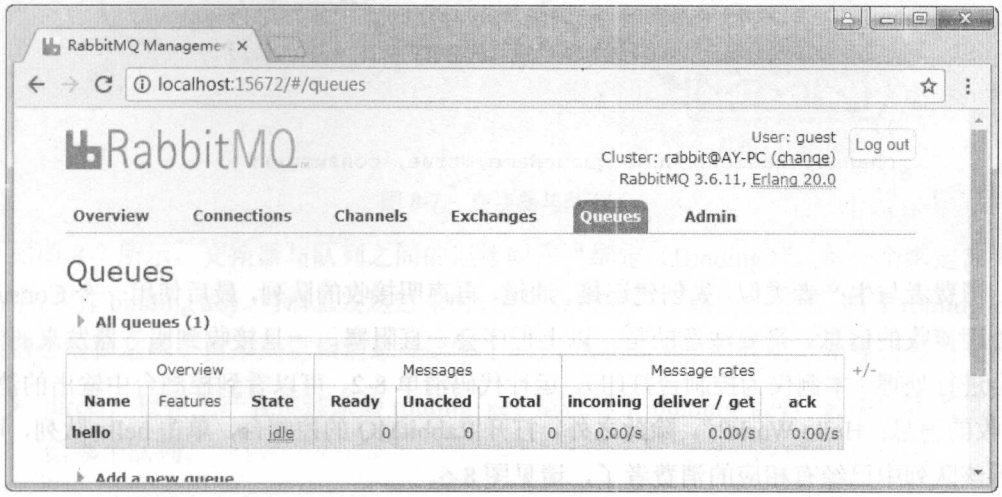


图 8-5 查看 RabbitMQ 队列

8.2.4 编写消费者

在本例中，生产者与消费者处于同一个项目，只是两个不同的运行类。消费者实现请见代码清单 8-2。

代码清单 8-2：codes\08\8.2\rabbit-test\src\main\java\org\crazyit\jms\Receive.java

```
public class Receive {

    public static void main(String[] argv) throws Exception {
        // 创建连接
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        Connection connection = factory.newConnection();
        // 创建通道
        Channel channel = connection.createChannel();
        // 声明队列
        String queueName = "hello";
        channel.queueDeclare(queueName, false, false, false, null);
        // 创建消费者
        Consumer consumer = new DefaultConsumer(channel) {
            @Override
```

```
            public void handleDelivery(String consumerTag, Envelope envelope,
                    AMQP.BasicProperties properties, byte[] body)
                    throws IOException {
                String message = new String(body, "UTF-8");
                System.out.println("接收的消息：" + message);
            }
        };
        channel.basicConsume(queueName, true, consumer);
    }
}
```

消费者与生产者类似，先创建连接、通道，再声明接收的队列，最后使用一个 Consumer 来处理接收的信息。需要注意的是，以上程序会一直阻塞，一旦接收到服务器发来的消息就会进行处理（本例仅为控制台打印）。运行代码清单 8-2，可以看到控制台中输出的消息："接收的消息：Hello World!"。除此之外，打开 RabbitMQ 的控制台，单击 hello 队列，可以看到该队列中已经有相应的消费者了，请见图 8-6。

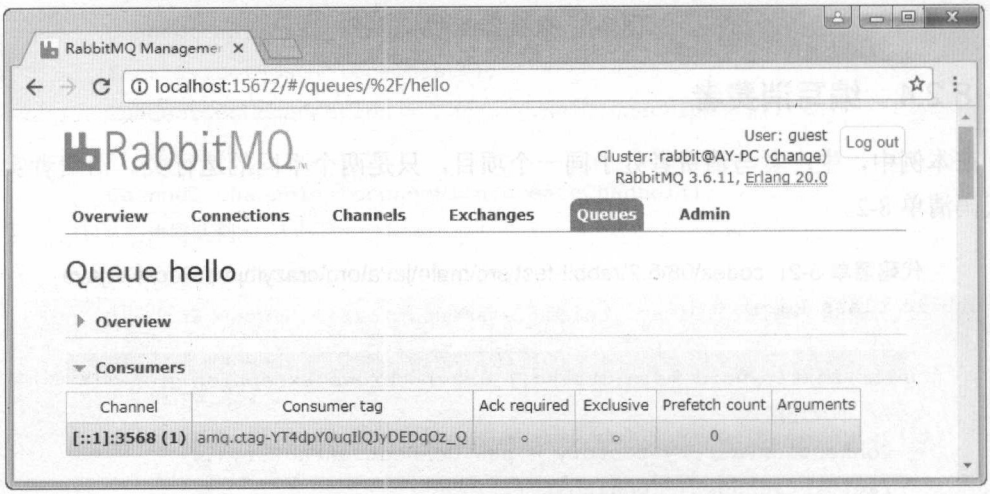


图 8-6　在控制台中查看消费者

到此，第一个 RabbitMQ 的程序已经完成，实现过程较为简单。下面，我将讲解一下 RabbitMQ 的几个重要模型。

8.2.5　交换器、绑定与队列

RabbitMQ 在设计上，生产者不会直接将消息发送给队列，它只会将消息发送给交换器，交换器一边从生产者接收消息，一边将消息发送给各个队列，交换器与队列请见图 8-7。

图 8-7 交换器与队列

如图 8-7 所示，交换器与队列之间的连接叫作“绑定（Binding）”。每一个绑定都有一个名字，叫作 binding key。当消息发送过来时，消息会带有一个路由的标识，叫作 routing key。交换器会根据这两个值来决定将消息发给哪个队列。RabbitMQ 中提供了 4 种类型的交换器。

- direct：根据生产者传过来的 routing key 是否等于 binding key，来决定将消息发送给哪个队列。
- topic：根据生产者传过来的 routing key 是否匹配一定的表达式，来决定消息发送给哪个或者哪些队列。
- fanout：将消息发送给交换器知道的全部队列，这种交换器会忽略设置的 routing key。
- headers：根据消息的头信息来决定将消息发送给哪个队列。

根据以上描述不难看出，交换器实际上也充当着路由器的作用。

本节对 RabbitMQ 进行了简单描述，编写了一个简单的 Hello World 程序。通过本节的内容，读者可以对 RabbitMQ 有一个大概了解，最重要的是理解消息代理的作用。关于 RabbitMQ 更加高级的应用不在本书讨论范围之内。接下来，将讲解 Spring Cloud Stream 支持的第二个消息中间件：Kafka。

8.3 Apache Kafka 框架

本节将对 Kafka 做一个简单的描述。

8.3.1 关于 Kafka

Kafka 是 Apache 下的一个用于处理数据流的分布式消息框架，它拥有水平扩展、容错、高效等特性，可以使用该框架实现以下功能：

- 构建在系统间进行实时数据传输的通道。
- 构建对数据流进行转换或响应的实时应用。

Kafka 的整体结构与 RabbitMQ 类似，消息生产者向 Kafka 服务器发送消息，Kafka 接收消息后，再投递给消费者。在 Kafka 中，生产者的消息会被发送到 Topic 中，Topic 中保存着各类数据，每一条数据都使用键、值进行保存。每一个 Topic 中都包含一个或多个物理分区（Partition），这些分区维护着消息的内容和索引，它们有可能被保存在不同的服务器中。对于客户端来说，无须关心数据如何被保存，只需关心将消息发往哪个 Topic。

8.3.2 运行 Kafka 服务器

Kafka 依赖了 ZooKeeper，启动 Kafka 服务器前，要先启动 ZooKeeper。本章所使用的 ZooKeeper 的版本为 3.4.8，Kafka 的版本为 2.11。下载两个框架的压缩包后进行解压，分别得到 zookeeper-3.4.8 与 kafka_2.11-0.11.0.0 目录。

先进入 zookeeper-3.4.8/conf 目录，将 zoo_sample.cfg 文件复制一份，并重命名为 zoo.cfg。使用命令行工具进入 zookeeper-3.4.8/bin 目录，运行 zkServer 命令，如果正常启动，将会占用 2181 端口，命令行窗口不必关闭，接下来启动 Kafka。

使用命令行工具进入 kafka_2.11-0.11.0.0/bin/windows 目录，运行 kafka-server-start ../../config/server.properties 命令启动 Kafka 服务器，如果正常启动，将会占用 9092 端口。此处的 Kafka 相当于前面章节中的 RabbitMQ 服务器，Kafka 同样提供了 API 让我们编写客户端。接下来，我们按照同样的方式使用 Kafka 的 API 来进行测试。

8.3.3 编写生产者

新建一个名称为 kafka-test 的 Maven 项目，加入以下依赖：

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.11.0.0</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.9</version>
</dependency>
```

新建生产者的运行类，请见代码清单 8-3。

代码清单 8-3：codes\08\8.3\kafka-test\src\main\java\org\crazyit\cloud\ProducerMain.java

```
public class ProducerMain {

    public static void main(String[] args) throws Exception {
        // 配置信息
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        // 设置数据 key 的序列化处理类
        props.put("key.serializer",
                "org.apache.kafka.common.serialization.StringSerializer");
        // 设置数据 value 的序列化处理类
        props.put("value.serializer",
                "org.apache.kafka.common.serialization.StringSerializer");
        // 创建生产者实例
        Producer<String, String> producer = new KafkaProducer<>(props);
        // 创建一条新的记录，第一个参数为 Topic 名称
        ProducerRecord record = new ProducerRecord<String, String>("my-topic",
            "userName", "Angus");
        // 发送记录
        producer.send(record);
        producer.close();
    }
}
```

生产者的代码较 RabbitMQ 的简单。创建属性实例，直接使用配置实例创建 Producer（生产者），再创建一个 ProducerRecord（记录），然后直接发送。在创建记录时，指定了向 my-topic 投递消息，消息的 key 为 userName，value 为 Angus。消息发送后，Kafka 会在服务器上创建一个相应的 Topic。运行代码清单 8-3，将消息投递到 Kafka 服务器的 Topic 中，接下来可以使用命令查看服务器的 Topic。

使用命令行工具进入 kafka_2.11-0.11.0.0/bin/windows 目录，输入命令 kafka-topics --list --zookeeper localhost:2181，可看到当前 Kafka 服务器的 Topic，如图 8-8 所示。

如果想删除服务器上的 Topic，可使用 kafka-topics --delete --zookeeper localhost:2181 --topic my-topic 命令。但在默认情况下，执行该命令只是将 Topic 标记为删除，如果想真正删除 Topic，需要修改 config/server.properties 文件，加入 delete.topic.enable=true 配置。

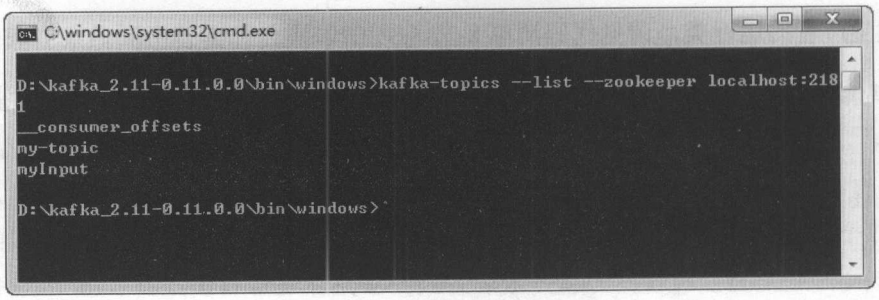


图 8-8 查看 Topic

8.3.4 编写消费者

本例中生产者与消费者同在一个项目，只是使用不同的启动类。前面小节在编写生产者时，指定消息发送到 my-topic，消费者订阅该 Topic 就可以获取到消息，详情请见代码清单 8-4。

代码清单 8-4：codes\08\8.3\kafka-test\src\main\java\org\crazyit\cloud\ConsumerMain.java

```
public class ConsumerMain {

    public static void main(String[] args) {
        // 配置信息
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        // 必须指定消费者组
        props.put("group.id", "test");
        props.put("key.deserializer",
                "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer",
                "org.apache.kafka.common.serialization.StringDeserializer");
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        // 订阅 my-topic 的消息
        consumer.subscribe(Arrays.asList("my-topic"));
        // 到服务器中读取记录
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(100);
            for (ConsumerRecord<String, String> record : records) {
                System.out.println("key: " + record.key() + ", value: " +
                    record.value());
            }
```

```
        }
    }
}
```

设置了配置信息后，创建一个 KafkaConsumer 实例，通过该实例订阅 my-topic 的消息，最后使用 KafkaConsumer 的 poll 方法获取服务器消息并输出。运行代码清单 8-3，再运行代码清单 8-4，可以看到输出如下：

```
key: userName, value: Angus
```

8.3.5 消费者组

在编写消费者时，需要指定消费者组的 id。关于消费者组，由于 Spring Cloud Stream 中也涉及这个概念，因此需要特别说明一下。

消费者会为自己添加一个消费者组的标识，每一条发布到 Topic 的记录，都会被交付给消费者组的一个消费者实例。如果多个消费者实例拥有相同的消费者组，那么这些记录将会分配到各个消费者实例上，以达到负载均衡的目的。如果所有的消费者都有不同的消费者组，那么每一条记录都会被广播到全部的消费者进行处理。如果理解不了这段文字，请见图 8-9。

图 8-9 消费者组

如图 8-9 所示，如果消费者 A 与消费者 B 属于同一个消费者组，那么当生产者发送过来一条消息时，仅会交给其中一个消费者处理；如果两个消费者不属于同一个消费者组，那么该消息会发给他们（广播）每一个进行处理。

接下来，将讲述在微服务中开发消息应用。

8.4 开发消息微服务

本节主要讲解如何在微服务中开发消息应用。根据前面章节的介绍，Spring Cloud

Stream 帮我们做了一定程度的简化，只需通过少量的代码配置就可以实现前面两个框架的功能，而不需要调用它们的 API。

8.4.1 准备工作

我们需要在微服务客户端中实现消息生产者与消费者，先建立以下几个项目。

- spring-server：Eureka 服务器，端口为 8761，代码目录为 codes\08\8.4\spring-server。
- spring-consumer：消息消费者，Eureka 客户端，注册到 Eureka，端口为 8080，代码目录为 codes\08\8.4\spring-consumer。
- spring-producer：消息生产者，Eureka 客户端，注册到 Eureka，端口为 8081，代码目录为 codes\08\8.4\spring-producer。

整个集群加上消息代理，结构如图 8-10 所示。

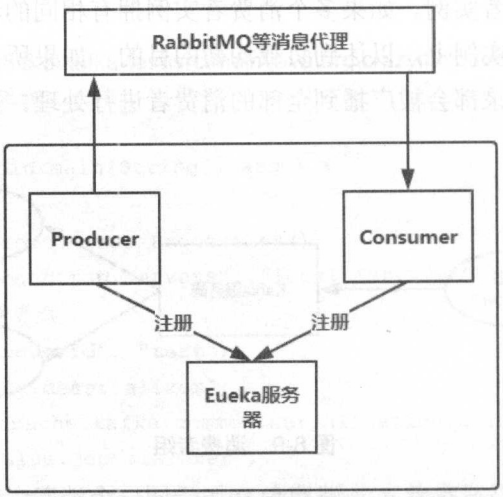


图 8-10 程序结构

由于 Spring Cloud Stream 帮我们实现了与消息代理交互的功能，因此对于集群中的生产者与消费者来说，不需要关心外部使用的是哪一个消息框架。本小节的案例默认使用 RabbitMQ，默认情况下，会连接本地的 5762 端口。如果需要在微服务中修改 RabbitMQ 的连接信息，可使用以下配置：

```
spring:
  application:
    name: spring-producer
```

```
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

8.4.2 编写生产者

在 spring-producer 项目中，加入以下依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

主要引入 spring-cloud-starter-stream-rabbit 依赖，该依赖会自动帮我们的项目引入 spring-cloud-stream 以及 spring-cloud-stream-binder。

接下来，编写发送服务，请见代码清单 8-5。

代码清单 8-5：codes\08\8.4\spring-producer\src\main\java\org\crazyit\cloud\SendService.java

```
public interface SendService {

    @Output("myInput")
    SubscribableChannel sendOrder();
}
```

新建一个 SendService 接口，添加 sendOrder 方法，该方法使用@Output 注解进行修饰。使用该注解表示会创建 myInput 的消息通道。调用该方法后，会向 myInput 通道投递消息。如果@Output 注解不提供参数，则使用方法名作为通道名称。接下来，需要让 Spring 容器开启绑定的功能，在 Application 类中，加入@EnableBinding 注解，请见代码清单 8-6。

代码清单 8-6：codes\08\8.4\spring-producer\src\main\java\org\crazyit\cloud\ProducerApplication.java

```
@SpringBootApplication
@EnableEurekaClient
@EnableBinding(SendService.class)
public class ProducerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProducerApplication.class, args);
    }
}
```

在@EnableBinding 注解中，以 SendService.class 作为参数，Spring 容器在启动时，会自动绑定 SendService 接口中定义的通道。编写控制器，调用 SendService 的发送方法，往服务器发送消息，请见代码清单 8-7。

代码清单 8-7：codes\08\8.4\spring-producer\src\main\java\org\crazyit\cloud\ProducerController.java

```
@RestController
public class ProducerController {

    @Autowired
    SendService sendService;

    @RequestMapping(value = "/send", method = RequestMethod.GET)
    public String sendRequest() {
        // 创建消息
        Message msg = MessageBuilder.withPayload("Hello World".getBytes()).build();
        // 发送消息
        sendService.sendOrder().send(msg);
        return "SUCCESS";
    }
}
```

在控制器中，通过@Autowired 自动注入 SendService，调用 sendOrder 方法得到 SubscribableChannel（通道）实例，再调用 send 方法，将 Hello World 字符串发送至消息代理中，本例默认的消息代理为 RabbitMQ。下面，先实现消息者，再一起整合测试。

8.4.3 编写消费者

消费者项目（spring-consumer）所使用的依赖与生产者一致，先编写接收消息的通道

接口，请见代码清单 8-8。

代码清单 8-8：codes\08\8.4\spring-consumer\src\main\java\org\crazyit\cloud\ReceiveService.java

```
public interface ReceiveService {

    @Input("myInput")
    SubscribableChannel myInput();
}
```

在 ReceiveService 中定义了一个 myInput 的消息输入通道，接下来，与生产者一样，在启动类中绑定消息通道，请见代码清单 8-9。

代码清单 8-9：codes\08\8.4\spring-consumer\src\main\java\org\crazyit\cloud\ReceiverApplication.java

```
@SpringBootApplication
@EnableBinding(ReceiveService.class)
public class ReceiverApplication {

    public static void main(String[] args) {
        SpringApplication.run(ReceiverApplication.class, args);
    }

    @StreamListener("myInput")
    public void receive(byte[] msg) {
        System.out.println("接收到的消息:  " + new String(msg));
    }
}
```

在启动类中同样使用了@EnableBinding 来开启绑定，并声明了通道的接口类。新建了一个 receive 方法，使用@StreamListener 注解进行修饰，声明了订阅 myInput 通道的消息。

依次启动 spring-server（8761 端口）、spring-producer（8081 端口）、spring-consumer（8080 端口），访问 http://localhost:8081/send，再打开消息者控制台，可以看到 Hello World 字符串的输出，证明消费者已经可以从消息代理中获取到消息。

8.4.4 更换绑定器

在前面的例子中，使用了 RabbitMQ 作为消息代理，如果想使用 Kafka，可以更换 Maven 依赖来实现。在生产者与消费者的 pom.xml 中，将 spring-cloud-starter-stream-rabbit 的依赖修改为 spring-cloud-starter-stream-kafka，请见以下配置：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

Spring Cloud 提供了绑定器的 API，目前实现了 RabbitMQ 与 Kafka 的绑定器。在笔者看来，绑定器更像适配器，对于我们的消息程序来说，并不关心使用了哪个消息代理，这些都由绑定器去实现。

8.4.5 Sink、Source 与 Processor

为了能简化开发，Spring Cloud Stream 内置了 3 个接口：Sink、Source 与 Processor，其中 Sink 接口的定义如下：

```
public interface Sink {

    String INPUT = "input";

    @Input(Sink.INPUT)
    SubscribableChannel input();
}
```

Source 接口的定义如下：

```
public interface Source {

    String OUTPUT = "output";

    @Output(Source.OUTPUT)
    MessageChannel output();
}
```

根据两个接口可知，实际上帮我们内置了 input 与 output 两个通道。那么在大多数情况下，我们就可以不必编写服务接口，甚至不必使用@Input 或@Output 两个注解。以消费者为例，在绑定通道时加入 Sink.class，请见以下代码片断：

```
@SpringBootApplication
@EnableBinding(value = {ReceiveService.class, Sink.class})
public class ReceiverApplication {

    @StreamListener(Sink.INPUT)
    public void receiveInput(byte[] msg) {
```

```
        System.out.println("receiveInput 方法，接收到的消息：  " + new String(msg));
    }
}
```

Processor 接口继承于 Sink 与 Source，在实际应用中，可以考虑只使用 Processor。

8.4.6 消费者组

在前面讲解 Kafka 框架的时候，提到了消费者组的概念。在 Spring Cloud Stream 中，同样引入这个概念。当消费者组相同时，对于发送过来的消息，仅由其中一个消费者实例处理，如果消费者组不同，则会发送给全部的消费者实例。

新建两个消费者项目，用于测试。

- spring-second-consumer：端口为 8082，在消息监听器中会输出 Second Consumer 等字符，代码目录为 codes\08\8.4\spring-second-consumer。
- spring-third-consumer：端口为 8083，在消息监听器中会输出 Third Consumer 等字符，代码目录为 codes\08\8.4\spring-third-consumer。

为原来的消费者项目（spring-consumer）配置消费者组的属性，在配置文件中加入以下属性：spring.cloud.stream.bindings.**myInput**.group.groupA，表示在 myInput 通道中，该消费者所属组为 groupA。使用同样的配置方式为新加的两个消费者项目配置它们的消息者组为 groupB。

生产者与消费者的大概结构请见图 8-11。

如图 8-11 所示，原来的消费者单独属于 groupA 组，新加入的两个消费者属于另外一个组。按照以下步骤启动程序：spring-server（8761）、spring-producer（8081）、spring-consumer（8080）、spring-second-consumer（8082）、spring-third-consumer（8083）。

启动后，访问生产者的发送地址 http://localhost:8081/send。在多次发送后可以看到，每一次发送的消息，原来的消费者（spring-consumer）都会接收到，而后面加的两个消费者，则会进行轮询处理。

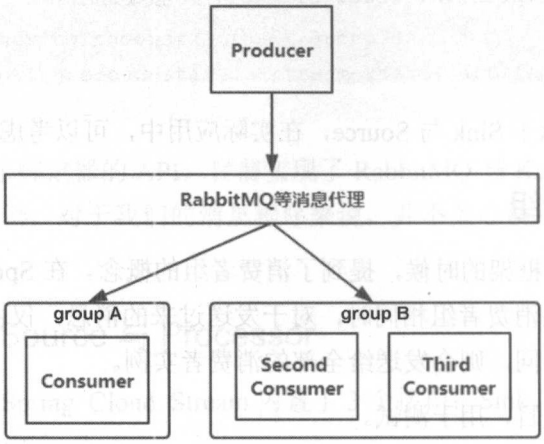


图 8-11 生产者与消费者的结构

8.5 本章小结

本章主要讲述了在微服务中开发消息应用，由于 Spring Cloud Stream 主要支持 RabbitMQ 与 Kafka 两个框架，因此在本章开头，主要介绍了这两个框架的使用，以及一些基本的概念和机制。在 8.4 节，讲述了如何使用 Spring Cloud 所提供的几个注解来简化消息程序的开发。

学习完本章后，读者对于 RabbitMQ、Kafka、Spring Cloud Stream 框架的基础知识会有一个初步的了解，本章可作为大家学习消息驱动的入门篇章。除了本章所介绍的内容外，不论是 RabbitMQ、Kafka 还是 Spring Cloud Stream，所涉及的知识多如牛毛，但由于篇幅所限，不能为大家一一讲解，望读者见谅。

CHAPTER

9

第9章 集群配置中心

本章要点

- Spring Cloud Config 介绍
- 构建配置中心的例子
- Spring Cloud Config 的配置
- 整合使用

从前面章节的讲解我们可以知道，基本上每个微服务都是使用 application.properties（yml）进行配置的。在实际应用时，集群中会存在多个服务，每个服务都可能部署多个实例，项目开始运营后，如何对集群的配置进行管理？如何实现修改配置而不用重启服务？这些问题都影响着集群的稳定。假如集群的服务节点配置混乱、修改常规配置还需要重启服务，这将直接增加运维人员的工作量，也给服务集群的稳定性带来了潜在的威胁。

Spring Cloud 已经为这些问题提供了解决方案：Spring Cloud Config，本章将以 Spring Cloud Config 为基础，讲述微服务集群中的配置。

9.1 概述

先了解一下 Spring Cloud Config 项目以及搭建环境。

9.1.1 关于 Spring Cloud Config

Spring Cloud Config 为分布式系统提供了配置服务器（简称服务器）和配置客户端（简称客户端），通过对它们的配置，可以很好地管理集群中的配置文件。在实际应用时，我们会将配置文件存放到外部系统（例如 Git、SVN 等），Spring Cloud Config 的服务器与客户端会到这些外部系统中读取、使用这些配置。

配置服务器主要有以下功能：

- 提供访问配置的服务接口。
- 对属性进行加密和解密。
- 可以简单地嵌入 Spring Boot 的应用中。

配置客户端主要有以下功能：

- 绑定配置服务器，使用远程的属性来初始化 Spring 容器。
- 对属性进行加密和解密。
- 属性改变时，可以对它们进行重新加载。
- 提供了与配置相关的几个管理端点。
- 在初始化引导程序的上下文时，进行绑定配置服务器和属性解密等工作，当然，也可以实现其他工作。

Spring Cloud 的程序可以通过简单地加入依赖，让其成为一个配置客户端，换言之，集

群中的各个节点，理论上都可以成为客户端。

9.1.2 应用结构

前面提到了 Spring Cloud Config 的配置服务器与配置客户端，在此，先以图形的方式展示一下它们的结构，请见图 9-1。

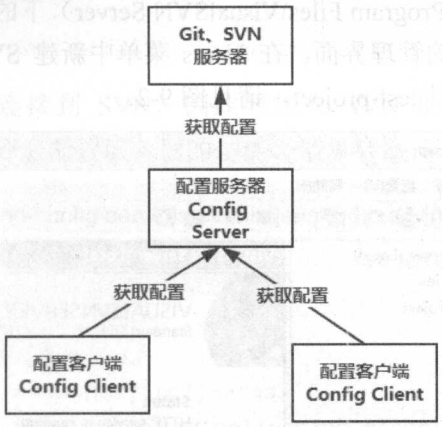


图 9-1 服务器与客户端的结构图

如图 9-1 所示，使用 Git、SVN 等版本的管理系统用于存放配置文件，服务器会到版本管理系统中获取配置，集群中的各个客户端（服务实例）再到服务器中获取配置。

9.1.3 引导程序简介

Spring Cloud 的程序在进行容器初始化时会先建立一个“引导上下文”（Bootstrap Context），再创建主应用的上下文。我们的主应用程序上下文通常读取的是 application.yml（或.properties）文件，而引导上下文则会读取 bootstrap.yml（或.properties）文件。因为 application.yml 的配置会在 bootstrap.yml 后加载，所以如果两份配置文件同时存在，且存在 key 相同的配置，则 application.yml 的配置会覆盖 bootstrap.yml 的配置。

配置客户端（Config Client）的引导程序在进行引导上下文创建时，会去读取外部的属性，并且会进行属性解密等工作。在此，大家对引导程序有以下两个认识即可：

- 引导上下文会在主应用上下文前创建，是主应用上下文的父上下文。
- 默认情况下，配置客户端的引导上下文在创建时会读取远程配置（去配置服务器中读取）。

9.1.4 搭建 SVN 环境

Spring Cloud Config 的服务器默认使用 Git 来管理配置内容，笔者就职的公司大部分都使用 SVN，因此本章也选用 SVN 作为配置管理工具。如果读者已经学会如何搭建 SVN 环境，或者有现成的环境，可以跳过本节。

本书所使用的 SVN 服务器为 64 位的 VisualSVN Server，版本为 3.6.4。安装完成后，进入安装目录（本例为 C:\Program Files\VisualSVN Server）下的 bin，单击启动 VisualSVN Server.msc，可进入 SVN 的管理界面。在 Users 菜单中新建 SVN 的用户并设置密码，在 Repositories 菜单中新建项目 test-project，请见图 9-2。

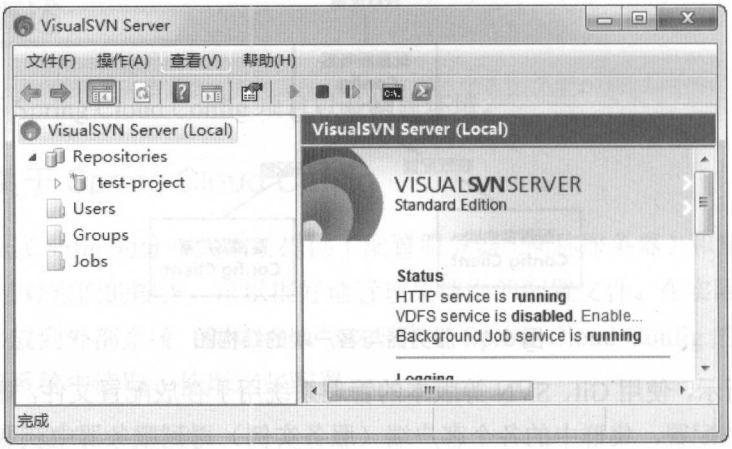


图 9-2 SVN 管理界面

接下来，安装 SVN 客户端，本节使用 TortoiseSVN，版本为 1.9.7。安装完成后，可以将前面新建的 test-project 项目检出（checkout）。

9.2 构建第一个例子

说一千道一万，不如举一个例子实际。本节将构建一个简单的例子，演示 Spring Cloud Config 的几个重要功能。

9.2.1 创建服务器

新建名称为 spring-config-server 的 Maven 项目，加入以下依赖：

```
<dependencies>
    <dependency>
```

```
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
    </dependency>
    <dependency>
        <groupId>org.tmatesoft.svnkit</groupId>
        <artifactId>svnkit</artifactId>
        <version>1.9.0</version>
    </dependency>
</dependencies>
```

由于配置服务器需要连接到 SVN，因此要加入 SVN 的依赖。新建启动类，使用@EnableConfigServer 注解开启配置服务器的功能，请见代码清单 9-1。

代码清单 9-1：codes\09\spring-config-server\src\main\java\org\crazyit\cloud\ConfigApplication.java

```
@SpringBootApplication
@EnableConfigServer
public class ConfigApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConfigApplication.class, args);
    }
}
```

简单两步，我们的项目就成为一个配置服务器了。接下来，让配置服务器连接到 SVN 仓库读取配置文件。

9.2.2 配置 SVN 仓库

为配置服务器（spring-config-server）的 application.yml 添加以下配置：

```
server:
  port: 8888
management:
  security:
    enabled: false
spring:
  profiles:
    active: subversion
  cloud:
    config:
      server:
```

```
        svn:
          uri: https://localhost/svn/test-project
          username: admin
          password: 123456
        default-label: default-config
```

配置服务的端口为8888，默认情况下，很多端点需要认证才能访问，配置management.security.enabled=false 关闭认证。为了能让配置服务器连上 SVN，需要先使用名称为subversion 的配置。spring-cloud-config-server 项目提供了 4 种配置，可以通过设置不同的名字来激活。

- git：默认值，表示去 Git 仓库读取配置文件。
- subversion：表示去 SVN 仓库读取配置文件。
- native：将去本地的文件系统中读取配置文件。
- vault：去 Vault 中读取配置文件，Vault 是一款资源控制工具，可对资源实现安全访问。

为 spring.profiles.active 配置以上 4 个值，可让配置服务器访问不同的仓库。在以上的配置中，指定了 SVN 仓库为 test-project，默认情况下，将会到 test-project/trunk 下获取配置文件。可以添加 spring.cloud.config.server.default-label 配置来指定获取的目录，以上配置指定了 default-config 为默认的配置目录。

为了能进行测试，在 SVN 中新建 default-config 目录，以及一份 first-test.yml 文件，如图 9-3 所示。

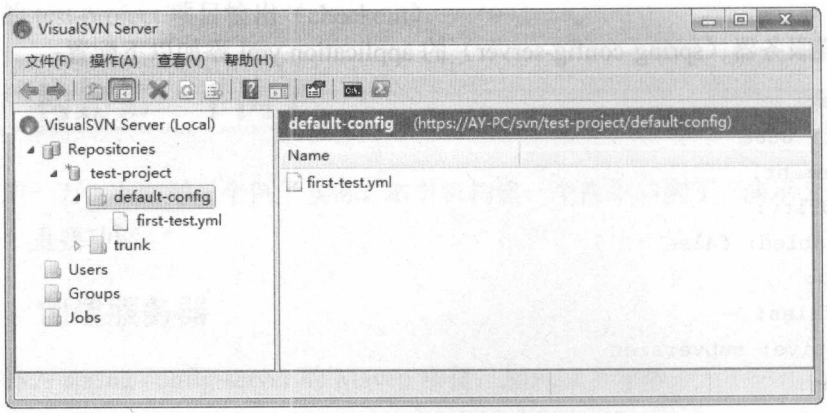


图 9-3 测试文件

测试文件 first-test.yml 配置了一个简单的属性 test.user.name=Angus。运行 spring-config-server 的启动类，用浏览器访问地址 http://localhost:8888/first-test.yml，可以看到输出如下：

```
test:
  user:
    name: Angus
```

接下来，编写客户端，测试获取配置文件。

9.2.3 创建客户端

新建名称为 spring-book-service 的 Maven 项目，加入以下依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>1.5.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
    <version>1.5.4.RELEASE</version>
</dependency>
```

建立启动类，并读取配置文件中的属性，请见代码清单 9-2。

代码清单 9-2：codes\09\spring-book-service\src\main\java\org\crazyit\cloud\BookApplication.java

```
@SpringBootApplication
@RestController
public class BookApplication {

    @Autowired
    private Environment env;

    @RequestMapping("/")
    public String home() {
        System.out.println("读取的值：" + env.getProperty("test.user.name"));
```

```
        return "Hello ";
    }

    public static void main(String[] args) {
        new
SpringApplicationBuilder(BookApplication.class).web(true).run(args);
    }
}
```

本例的启动类还承担了 Controller 的工作。当访问应用主页时，会在控制台输出 test.user.name 的属性值，而该值将会存在于 SVN 仓库的一份配置文件中。

9.2.4 从客户端读取 SVN 配置

由于客户端要在引导程序中读取配置服务器的配置，因此要新建名为 bootstrap.yml 的配置文件。配置文件内容请见代码清单 9-3。

代码清单 9-3：codes\09\spring-book-service\src\main\resources\bootstrap.yml

```
spring:
  application:
    name: spring-book-service
  cloud:
    config:
      uri: http://localhost:8888
      profile: dev
management:
  security:
    enabled: false
```

在代码清单 9-3 的配置中，设定了应用名称为 spring-book-service，使用 spring.cloud.config.uri 来设定配置服务器的位置，而且使用 spring.cloud.config.profile 来指定读取的配置。最终，配置客户端会到 SVN 服务器的 test-project/default-config 目录下读取 spring-book-service-dev.yml（.properties）。以下的配置同样会读取 spring-book-service-dev.yml：

```
spring:
  cloud:
    config:
      uri: http://localhost:8888
      profile: dev
      name: spring-book-service
```

以上配置使用了 spring.cloud.config.name 来代替 spring.application.name。如果两个属性都不提供，那么默认情况下会去读取 application-dev.yml 文件。

新建一份 spring-book-service-dev.yml 文件，上传到 SVN 的 test-project/default-config，该文件的内容如下：

```
test:
  user:
    name: Angus
```

以上配置的值将会被客户端远程读取。依次启动服务器（8888 端口）和客户端（8080 端口），在浏览器中访问 http://localhost:8080/，可以看到客户端的控制台输出如下：

```
读取的值：Angus
```

根据结果可知，属性值已经被读取。我们建立的这个客户端，默认会到 SVN 的 test-project/default-config 下读取配置。如果客户端想指定读取某个分支版本，或者其他目录下的配置，可以设置 spring.cloud.config.label 来覆盖服务器的 default-label 属性，例如以下配置：

```
spring:
  application:
    name: spring-book-service
  cloud:
    config:
      uri: http://localhost:8888
      profile: dev
      label: book-module
```

在以上的配置中，该客户端会到 test-project/book-module 下读取 spring-book-service-dev.yml。

9.2.5 目录配置总结

前面的小节对服务器与客户端的配置进行了简单的演示，如果读者对相关的目录配置有所疑问，可以阅读本小节。配置服务器和客户端，主要有与以下几个目录相关的配置。

- 服务器：spring.cloud.config.server.svn.uri = https://localhost/svn/test-project
- 服务器：spring.cloud.config.server.default-label = default-config
- 客户端：spring.application.name = spring-book-service
- 客户端：spring.cloud.config.profile = dev

从以上的4个配置中不难看出，服务器的两个配置用于指定到SVN的路径、配置目录，而客户端的两个配置则用于指定读取哪份配置文件。根据以上配置，客户端将会去 https://localhost/svn/test-project/default-config 目录下读取名称为 spring-book-service-dev.yml 的文件。

如果客户端想指定目录，可以通过配置 spring.cloud.config.label = book-module 实现，配置后，客户端将会去 https://localhost/svn/test-project/book-module 下读取配置文件。

另外，在实际环境中有可能存在多份配置文件，例如一份文件是专门配置 Hystrix 的，另一份是专门配置 Zuul 的，可以配置为 spring.cloud.config.profile = hystrix, zuul，此时，将会去读取名称为 spring-book-service-hystrix.yml 与 spring-book-service-zuul.yml 的配置文件。

9.2.6 刷新配置

远程SVN服务器上面的配置修改后，需要通知客户端来改变配置。可以访问客户端的 /refresh 端点进行刷新，访问该端点要使用 HTTP 的 POST 方法。修改 SVN 上面的 spring-book-service-dev.yml，将 test.user.name 修改为 crazyit，再使用 HttpClient 来发送请求，刷新配置。发送 POST 请求的代码请见代码清单 9-4。

代码清单 9-4：codes\09\spring-book-service\src\main\java\org\crazyit\cloud\RefreshClient.java

```
// 创建默认的 HttpClient
CloseableHttpClient httpclient = HttpClients.createDefault();
// 调用 POST 方法请求服务
HttpPost httpMethod = new HttpPost("http://localhost:8080/refresh");
// 获取响应
HttpResponse response = httpclient.execute(httpMethod);
// 根据响应解析出字符串
System.out.println(EntityUtils.toString(response.getEntity()));
```

修改配置并提交到 SVN 服务器后再运行代码清单 9-4，可以看到控制台输出如下：

```
["test.user.name"]
```

客户端的 refresh 服务在接收到请求后，会重新到配置服务器获取最新的配置，然后用最新的配置与旧配置进行对比，最终将有修改的配置 key 返回给服务调用者。在整个过程中，配置服务器、客户端都不需要重新启动，调用 refresh 刷新配置后，可以再访问客户端来查看配置是否更新。在浏览器中访问 http://localhost:8080，可以看到客户端控制台输出为最新的值。

9.2.7 刷新 Bean

前面的章节中介绍了如何进行配置刷新，然而在实际应用中，往往不仅是只刷新一个配置的值那么简单。由于 Spring 容器中的很多 Bean 都是根据某个属性值来进行初始化的，配置一旦更新，需要重建这个 Bean 的实例。为了解决该问题，Spring Cloud 提供了@RefreshBean 注解。

在 Spring 的容器中，有一个类型为 RefreshBean 的 Bean。当/refresh 端点被访问时，负责处理刷新的 ContextRefresher 类会先去远程的配置服务器刷新配置，然后再调用 RefreshBean 的 refreshAll 方法来处理实例。容器中使用了@RefreshBean 注解进行修饰的 Bean，都会在缓存中进行销毁，当这些 Bean 被再次引用时，就会创建新的实例，以此达到一个“刷新”的效果。

在客户端（spring-book-service）中测试刷新 Bean，新建配置类，请见代码清单 9-5。

代码清单 9-5：codes\09\spring-book-service\src\main\java\org\crazyit\cloud\MyConfig.java

```
@Configuration
public class MyConfig {

    @Bean
    @RefreshScope
    public Person person(Environment env) {
        // 读取名字创建 Person 实例
        String name = env.getProperty("test.user.name");
        // 输出 Person 名字
        System.out.println("初始化 person bean: " + name);
        // 创建一个 Person
        Person p = new Person();
        p.setName(name);
        return p;
    }

    static class Person {

        private String name;

        public String getName() {
            return name;
        }
```

```
        public void setName(String name) {
            this.name = name;
        }
    }
}
```

在配置类中会创建一个 Person 的 Bean，该 Bean 会读取 test.user.name 属性来创建 Person 实例，该属性保存在 SVN 的配置文件中。修改一下控制器，请见代码清单 9-6。

代码清单 9-6：codes\09\spring-book-service\src\main\java\org\crazyit\cloud\BookApplication.java

```
@Autowired
private Person person;

@RequestMapping("/person")
public String getPerson() {
    System.out.println("输出 Person 实例: " + person + ", 名字: " + person.getName());
    return "Hello";
}
```

往控制器中注入 Person，访问/person 地址后，会输出当前容器中的 Person 以及名字。当前 SVN 中 spring-book-server-dev.yml 的 test.user.name 属性值为 crazyit，依次启动服务器、客户端，访问 http://localhost:8080/person，可以看到客户端的控制台输出如下：

```
初始化 person bean: crazyit
输出 Person 实例: org.crazyit.cloud.MyConfig$Person@1960b5b, 名字: crazyit
```

修改 SVN 中的配置文件并提交，然后向 http://localhost:8080/refresh 端点发送 POST 请求（运行 RefreshClient），最后再次访问 http://localhost:8080/person，可以看到客户端控制台输出如下：

```
初始化 person bean: Angus
输出 Person 实例: org.crazyit.cloud.MyConfig$Person@931c64, 名字: Angus
```

根据输出可知，属性值已经被刷新，使用该属性值创建的 Person 实例已经被重新创建。

9.3 配置的加密和解密

在实际应用中会涉及很多敏感的数据，这些数据会被加密保存到 SVN 仓库中，最常见的就是数据库密码。Spring Cloud Config 为这类敏感数据提供了加密和解密的功能，加密后的密文在传输给客户端前会进行解密。配置服务器支持对称加密和非对称加密，对称加密使用 AES 算法，非对称加密使用 RSA 算法。

9.3.1 为服务器安装 JCE

服务器的加密和解密依赖 JCE（Java Cryptography Extension），可到 Oracle 官方网站下载 Java 8 JCE，也可以直接在本书所附的 soft 目录下载，文件名为 jce_policy-8.zip。下载了 zip 包后，解压会得到 local_policy.jar 和 US_export_policy.jar 两个 jar 包，将其复制到 $JAVA_HOME/jre/lib/security 目录下，覆盖原来的两个 jar 包，即可完成安装。

9.3.2 加密和解密端点

本小节以对称加密为基础，介绍一下配置服务器提供的加密和解密端点，这两个端点可以当作工具来使用，方便我们进行加密和解密。修改配置服务器（spring-config-server）的 application.yml 文件，设置加密和解密的 key，请见代码清单 9-7。

代码清单 9-7：codes\09\spring-config-server\src\main\resources\application.yml

```
encrypt:
  key: myKey
```

加密的服务端点为/encrypt，需要向该端点发送 POST 请求。请求体中包含需要加密的明文，我们使用 HttpClient 来向该端点发送请求，请见代码清单 9-8。

代码清单 9-8：codes\09\spring-config-server\src\main\java\org\crazyit\cloud\EncryptClient.java

```
// 调用 POST 方法请求服务
HttpPost httpMethod = new HttpPost("http://localhost:8888/encrypt");
HttpEntity entity = new StringEntity("crazyit", Consts.UTF_8);
httpMethod.setEntity(entity);
// 获取响应
HttpResponse response = httpclient.execute(httpMethod);
// 根据响应解析出字符串
System.out.println(EntityUtils.toString(response.getEntity()));
```

在代码清单 9-8 中，向/encrypt 端点发送 POST 请求，请求为 crazyit 字符串进行加密。启动配置服务器（codes\09\spring-config-server，端口为 8888），再运行代码清单 9-8，控制台输出如下：

```
797e316ce5c1dc9ce02d6eca929ee48c577efd2e379d79171d454a9b816818cd
```

访问的/encrypt 端点会使用配置的 key 对明文进行加密并返回密文。接下来，测试解密端点/decrypt，同样使用 HttpClient 访问该端点，请见代码清单 9-9。

代码清单 9-9：codes\09\spring-config-server\src\main\java\org\crazyit\cloud\DecryptClient.java

```
// 调用 POST 方法请求服务
HttpPost httpMethod = new HttpPost("http://localhost:8888/decrypt");
HttpEntity entity = new StringEntity(
    "797e316ce5c1dc9ce02d6eca929ee48c577efd2e379d79171d454a9b816818cd",
        Consts.UTF_8);
httpMethod.setEntity(entity);
// 获取响应
HttpResponse response = httpclient.execute(httpMethod);
// 根据响应解析出字符串
System.out.println(EntityUtils.toString(response.getEntity()));
```

调用/decrypt 端点时，向其 POST 密文，用我们配置的 key 对密文进行解密，最后返回解密后的明文。

9.3.3 SVN 存储加密数据

在 SVN 仓库的 yml（properties）配置文件中，使用“{cipher}密文”的格式来保存加密后的数据。假设 MySQL 数据库的密码为 crazyit，服务器中配置的 key 为 myKey，对 crazyit 字符串加密后的密文为 797e316ce5c1dc9ce02d6eca929ee48c577efd2e379d79171d454a9b816818cd。为 SVN 仓库中的 test-project/default-config/spring-book-service-dev.yml 文件进行以下配置：

```
mysql:
  passwd: '{cipher}797e316ce5c1dc9ce02d6eca929ee48c577efd2e379d79171d454a9b816818cd'
```

启动配置服务器后，在浏览器中访问以下地址 http://localhost:8888/spring-book-service-dev.yml，浏览器的输出如下：

```
mysql:
  passwd: crazyit
test:
```

```
  user:
    name: Angus
```

可以看到，配置服务器已经对数据进行了解密。需要注意的是，如果使用的是 properties 文件，在配置时需要把单引号去掉。

9.3.4 非对称加密

对称加密算法在加密和解密时都使用同一个密钥，而非对称加密则使用一对密钥。使用公钥加密时，解密就需要使用私钥。Spring Cloud Config 同样支持非对称加密，使用 keytool 工具生成一对密钥用于加密和解密，在控制台输入以下命令：

```
keytool -genkeypair -alias "testKey" -keyalg "RSA" -keystore "D:\myTest. keystore"
```

按提示输入相关信息后，将生成的 myTest.keystore 复制到配置服务器 spring-config-server 的 classpath 下（本例与 application.yml 同目录），修改 application.yml 文件，加入以下内容：

```
encrypt:
  keyStore:
    location: classpath:/myTest.keystore          # keystore 位置
    password: crazyit                             # 密钥库的密码
    alias: testKey                                # 密钥对的别名
    secret: crazyit                               # 密钥口令
```

完成以上工作后，启动配置服务器，向 http://localhost:8888/encrypt（加密端点）发送 POST 请求，可以得到加密后的密文。密文的使用与前面的对称加密类似，在 SVN 的配置文件中同样使用“{cipher}密文”的格式（注意，yml 文件需要单引号），访问相应配置时，配置服务器会自动进行解密。

9.4 其他配置

前面章节介绍了 Spring Cloud Config 的主要配置，接下来对服务器与客户端的其他功能进行介绍。

9.4.1 服务器健康指示器

默认情况下，服务器会访问配置的 SVN 仓库 uri，如果连接不上，那么服务器的健康

状态将会置为DOWN。除了会检测SVN仓库的uri是否可以访问外，还可以进行额外配置，检测uri下面的目录是否可以连接，请见以下配置：

```
spring:
  profiles:
    active: subversion
  cloud:
    config:
      server:
        svn:
          uri: https://localhost/svn/test-project
          username: admin
          password: 123456
        default-label: default-config
        health:
          repositories:
            book-service:
              label: health-test
```

配置服务器会去SVN中连接https://localhost/svn/test-project/health-test，如果SVN上不存在该目录，在访问服务器的/health时，将会得到以下JSON字符串：

```
{
    "status": "DOWN",
    "configServer": {
        "status": "DOWN",
        "repository": {
            "application": "book-service",
            "profiles": "default"
        },
        "error": "org.springframework.cloud.config.server.environment.
            NoSuchLabelException: No label found for: health-test"
    }
}
```

如果不想启用健康指示器，可将spring.cloud.config.server.health.enabled属性设置为false。

9.4.2 客户端的错误提前与重试机制

在实际应用中可能会有一些特殊需求，例如客户端比较关心配置服务器是否能连接上，在启动时如果无法连接，宁可自己启动失败，也不能带着“错误”启动容器。对于这种情

况，可以使用 Spring Cloud Config 的错误提前机制。在客户端的 bootstrap.yml 中，将 spring.cloud.config.failFast 属性设置为 true，即可实现错误提前，只要在启动时无法连接（一次）配置服务器，则会中止容器的启动。

如果连接错误，就会有与之配套的重试机制。先将 spring.cloud.config.failFast 设置为 true，再为客户端添加 spring-retry 和 spring-boot-starter-aop 的依赖：

```
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
    <version>1.2.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
    <version>1.5.3.RELEASE</version>
</dependency>
```

完成以上两步后就可以在客户端的 bootstrap.yml 配置文件中设置重试的参数，主要使用以下几个参数。

- spring.cloud.config.retry.initial-interval：初始的重试间隔，默认为 1000 毫秒。
- spring.cloud.config.retry.max-attempts：最大重试次数，默认为 6。
- spring.cloud.config.retry.max-interval：最大的重试间隔，默认为 2000 毫秒。
- spring.cloud.config.retry.multiplier：重试间隔的递增系数，默认为 1.1。

9.4.3 安全配置

配置服务器可以使用 spring-boot-starter-security 模块来设置客户端的访问权限，在配置服务器中加入以下依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
    <version>1.5.3.RELEASE</version>
</dependency>
```

修改服务器的 application.yml，配置访问的用户名和密码：

```
security:
  user:
    name: root
    password: crazyit
```

进行以上配置后，客户端在启动或者获取配置时，就会得到 401 错误，为客户端的 bootstrap.yml 加入以下配置，即可解决权限问题：

```
spring:
  cloud:
    config:
      username: root
      password: crazyit
```

9.4.4 访问服务器配置

在前面的章节中，启动配置服务器后，可以使用 http://localhost:8888/spring-book-service-dev.yml 这样的地址来访问配置。下面对配置的访问规则进行简单描述。下面的规则可以访问 SVN 中的资源：

- /{application}/{profile}[/{label}]
- /{application}-{profile}.yml
- /{label}/{application}-{profile}.yml
- /{application}-{profile}.properties
- /{label}/{application}-{profile}.properties

以上的 HTTP 资源，application 表示客户端应用的名称，可使用 spring.application.name 进行配置，profile 表示客户端所使用的配置，label 表示所配置的目录标签。这 3 个属性在前面章节已经有过相应的描述，在此不再赘述。

9.5 整合使用

本节将讲述 Spring Cloud Config 与其他框架的整合，包括 Eureka、Zuul、Spring Cloud Bus 等。

9.5.1 准备工作

本节的案例主要涉及 5 个项目，这些项目都可以在 codes\09\9.5 目录中找到。这 5 个项

目的大概信息如下。

- eureka-server：Eureka 服务器，端口为 8761。
- eureka-config-server：配置服务器，同时也是 Eureka 客户端，端口为 8899。
- eureka-config-client：配置客户端，同时也是 Eureka 客户端，端口为 8081，在本例充当普通的服务实例角色。
- eureka-zuul：集群网关，既是 Eureka 客户端，也是配置客户端，会到配置服务器抓取路由规则，端口为 9000。
- eureka-bus：在本例中，它主要向消息中间件（RabbitMQ 等）发送消息，通知所有的节点更新配置，端口为 10000。

以上项目对应的结构请见图 9-4。

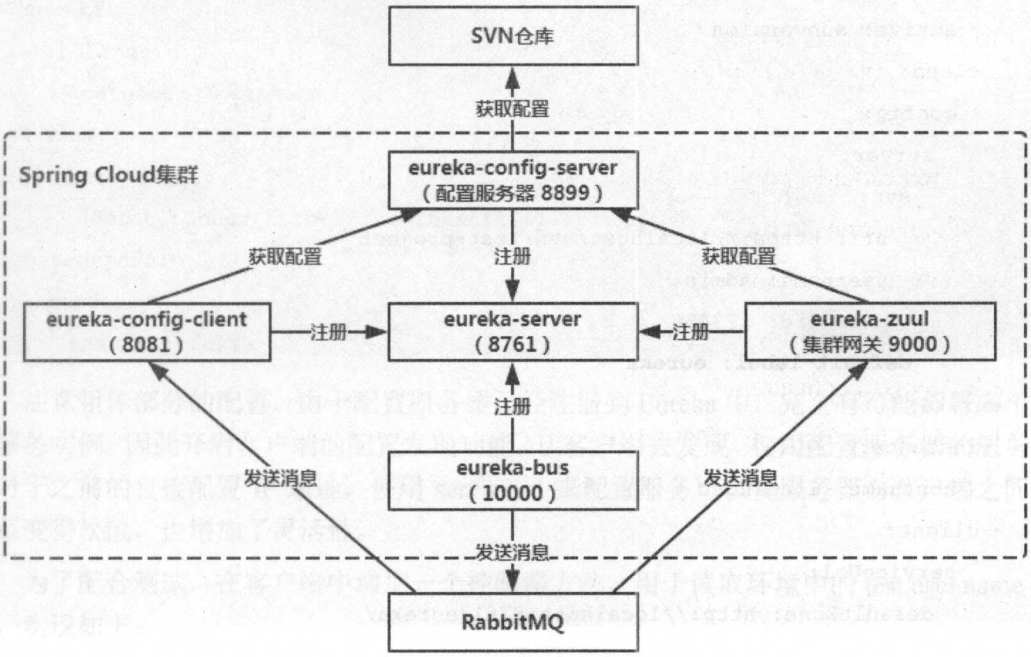


图 9-4 整体结构

图 9-4 展示了本例中的 5 个项目在集群中的职责及位置，项目名称下括号中的内容为对应项目的端口号。需要注意的是，eureka-zuul 项目既是配置客户端，也是集群的网关，它会去配置服务器获取路由配置。

9.5.2 配置服务器、客户端整合 Eureka

配置服务器为 eureka-config-server 项目，pom.xml 中有 spring-cloud-config-server、spring-cloud-starter-config、spring-cloud-starter-eureka、svnkit 这 4 个依赖，代码清单 9-10 为配置服务器的配置。

代码清单 9-10：codes\09\9.5\eureka-config-server\src\main\resources\application.yml

```
server:
  port: 8899
spring:
  application:
    name: eureka-config-server
  profiles:
    active: subversion
  cloud:
    config:
      server:
        svn:
          uri: https://localhost/svn/test-project
          username: admin
          password: 123456
        default-label: eureka
eureka:
  instance:
    hostname: localhost
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

配置服务器的 application.yml 与前面章节中的类似，仅加入了 eureka 注册的配置。接下来，对客户端（eureka-config-client）项目进行配置，该项目的 pom.xml 中引入了 spring-cloud-starter-config、spring-cloud-starter-eureka、spring-boot-starter-actuator、**spring-cloud-starter-bus-amqp** 这 4 个依赖。由于配置客户端会作为一个消息消费者接收 RabbitMQ 发送的消息，因此要使用 amqp 的依赖。该项目的 bootstrap.yml 文件，请见代码清单 9-11。

代码清单 9-11：codes\09\9.5\eureka-config-client\src\main\resources\bootstrap.yml

```
server:
  port: 8081
spring:
  application:
    name: eureka-config-client
  cloud:
    config:
      discovery:
        enabled: true
        service-id: eureka-config-server
      profile: dev
eureka:
  instance:
    hostname: localhost
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
management:
  security:
    enabled: false
```

注意粗体部分的配置。由于配置服务器已经注册到 Eureka 中，完全有可能部署多个配置服务实例，因此开启客户端的配置发现功能，让客户端去发现、使用配置服务器的服务。相对于之前的直接配置 IP 地址，使用 service-id 来配置服务可以使服务器与客户端之间的关系变得松散，也增加了灵活性。

为了配合测试，在客户端中增加一个控制器方法，用于读取环境中的 test.user.name 属性，实现如下：

```
@Autowired
private Environment env;

@RequestMapping("/")
public String home() {
    String name = env.getProperty("test.user.name");
```

```
    return "Hello " + name;
}
```

接下来，新建一份 eureka-config-client-dev.yml 配置文件，文件中只有 test.user.name 属性，值为 Angus，将其上传到 SVN 的 https://localhost/svn/test-project/eureka 目录。

依次启动 Eureka 服务器（8761）、配置服务器（8899）、配置客户端（8081），在浏览器中访问 http://localhost:8081，可以看到浏览器输出了属性值。根据前面章节的介绍可知，SVN 上面的配置文件改变后，可以调用客户端的/refresh 进行配置刷新。

如对 Eureka 服务器、配置服务器、配置客户端的结构有所疑问，可参见图 9-5。

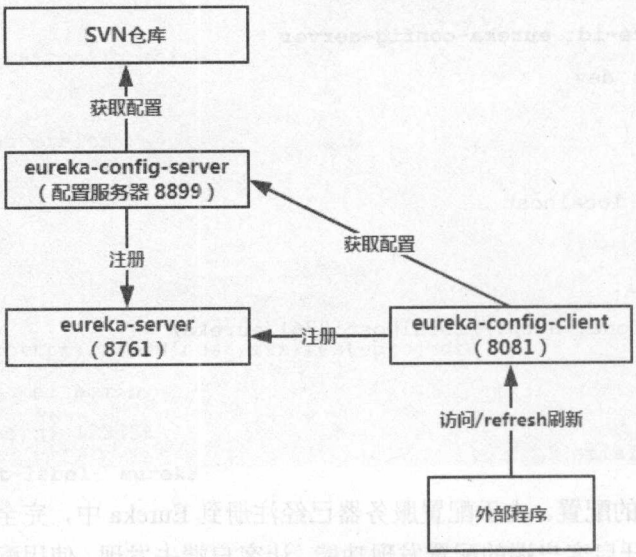


图 9-5 配置服务器与配置客户端

图 9-5 实际上是图 9-4 的部分截取。

9.5.3 整合 Zuul

如图 9-4 所示的结构图，eureka-zuul 项目实际上也是一个配置客户端，它会去配置服务器中获取路由配置。其取得路由配置后，调用配置客户端提供的/refresh 端点来刷新配置，这样就可以实现动态路由功能。

本例的 eureka-zuul 项目在 pom.xml 中加入了以下几个依赖：

- spring-cloud-starter-config

- spring-cloud-starter-eureka
- spring-cloud-starter-zuul
- spring-boot-starter-actuator
- spring-cloud-starter-bus-amqp

该项目的依赖基本与前面的配置客户端类似，只是多了一个 spring-cloud-starter-zuul。该项目的 bootstrap.yml 配置，请见代码清单 9-12。

代码清单 9-12：codes\09\9.5\eureka-zuul\src\main\resources\bootstrap.yml

```
server:
  port: 9000
spring:
  application:
    name: eureka-zuul
  cloud:
    config:
      discovery:
        enabled: true
        service-id: eureka-config-server
      profile: rel
eureka:
  instance:
    hostname: localhost
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
management:
  security:
    enabled: false
```

网关项目的配置基本与前面的配置客户端（eureka-config-client 项目）类似，同样使用 service-id 来查找配置服务器。细心的读者可能会发现，作为网关项目，配置中却没有任何路由配置。为了实现动态路由，该项目的路由配置都放到了 SVN 中。新建一份 eureka-zuul-rel.yml 配置文件，上传到 https://localhost/svn/test-project/eureka 目录，配置文件的内容如下：

```
zuul:
  routes:
```

```
routeTest:
  path: /routeTest
  url: http://www.sohu.com
```

当访问 http://localhost:9000/routeTest 时，会自动路由到 http://www.sohu.com。依次启动 Eureka 服务器、配置服务器、网关，访问 http://localhost:9000/routeTest，可以看到最终跳转到搜狐网站。修改 SVN 上面的配置文件，让其跳转到 163 网站，修改完成并上传 SVN 后，向 http://localhost:9000/refresh 端点发送 POST 请求刷新配置。完成后再次访问/routeTest，成功跳转到 163 网站。在整个过程中都不需要对任何节点进行重启操作，只需要访问/refresh 端点即可完成。

9.5.4 整合 Spring Cloud Bus 刷新配置

SVN 仓库上面的配置更新后，如果要刷新配置，前面需要调用服务的/refresh 端点。如果集群中存在多个要刷新的节点，逐个去刷新配置将会产生巨大的工作量。要解决该问题，可以配合 Spring Cloud Bus 来实现全部配置的刷新。先了解一下结构图，请见图 9-6。

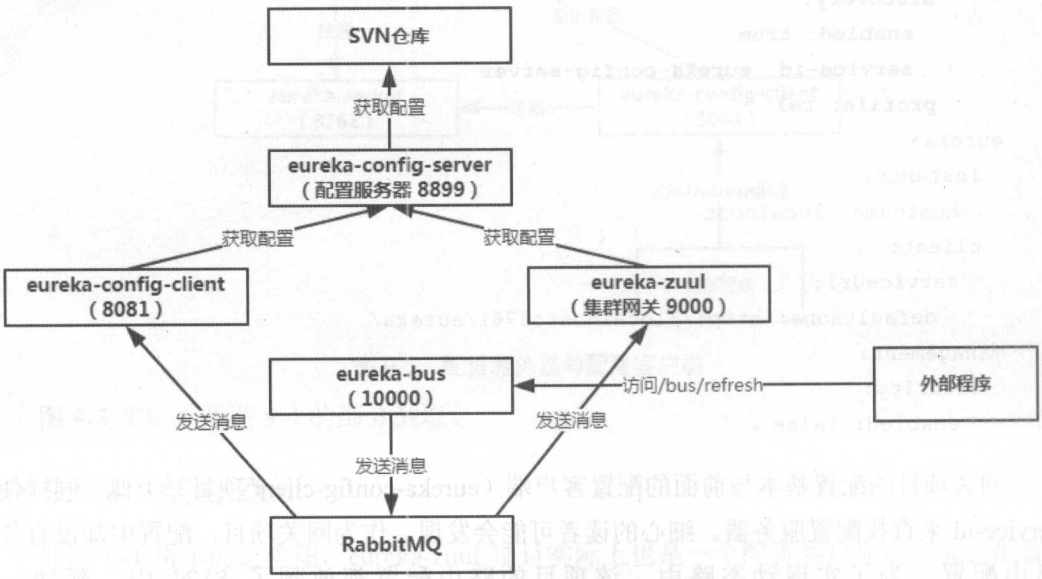


图 9-6 刷新集群配置

图 9-6 实际上是图 9-4 的截取，外部程序调用 eureka-bus 的/bus/refresh 端点后，eureka-bus 发送消息到 RabbitMQ，RabbitMQ 将消息广播给各个消息消费者，也就是集群中的 eureka-config-client 和 eureka-zuul，这些消息客户端接收到消息后，会重新加载程序的配置，

效果好似逐个调用了/refresh 端点。

要实现以上效果，eureka-config-client 和 eureka-zuul 两个配置客户端要加入 spring-cloud-starter-bus-amqp 的依赖（前面已讲述）。除此之外，发送消息的 eureka-bus 项目，使用以下依赖：

- spring-cloud-starter-bus-amqp
- spring-cloud-starter-config
- spring-cloud-starter-eureka

默认情况下会连接本地的5672端口来连接RabbitMQ，如果需要连接远程的RabbitMQ，可使用以下配置：

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

9.5.5 刷新单个节点配置

在一些情况下，SVN 更新了多个项目的配置，但调用者只希望刷新其中一个项目的配置。要实现这样的需求，可以直接访问该项目的/refresh 端点。但遗憾的是，在实际环境中，集群中的节点有可能根本不对外开放这些端点，此时，可以为“总线”项目的/bus/refresh 端点添加 destination 请求参数，刷新指定项目配置。例如可以访问以下地址，刷新网关实例的配置/bus/refresh?destination=eureka-zuul:9000，参数值的格式为“服务 ID:端口”，如果想刷新该服务的全部实例，可以使用 eureka-zuul:**这样的格式。

9.6 本章小结

本章主要以 Spring Cloud Config 为基础，讲述了如何搭建微服务的配置中心。在本章开头，先构建了一个基本的例子向大家展示了 Spring Cloud Config 最主要的几个功能，例如 SVN 仓库配置、远程文件读取、配置刷新等内容。学习完 9.2 节后，读者基本上可以上手使用 Spring Cloud Config。在 9.3 节，讲解了如何对配置进行加密和解密，9.4 节讲解了一些常用配置。

在本章的9.5节，主要涉及Spring Cloud Config与其他框架的整合使用，包括与Zuul整合实现动态路由的功能，与Spring Cloud Bus整合实现集群的配置刷新功能。掌握这些重要的功能，对提升集群的稳定性、减少集群的维护工作量，都有非常重要的意义。

CHAPTER

10

第10章 微服务跟踪

本章要点

- 服务跟踪的意义以及 Sleuth 框架
- Sleuth 与 Zipkin 整合
- Sleuth 与 ELK 整合

Spring Cloud 提供了一个 Sleuth 框架，用于跟踪微服务的调用过程，本章将以该框架为核心，讲述微服务跟踪的相关知识。

10.1 概述

先对微服务跟踪的相关概念做一个基本的讲解。

10.1.1 实际问题与 Sleuth

在前面的章节中，我们使用 Spring Cloud 来搭建服务集群，不论是 Eureka 服务器、服务实例，还是配置服务器、网关等节点，都可以横向扩展。一旦集群中的服务数量增多，并且它们之间存在复杂的依赖关系，那么管理它们将会变成一件很棘手的事情。

当外部用户向集群发起请求时，这些请求将会调用多个服务，每个服务又会依赖其他的服务，此时，如果出现异常、超时等情况，排查问题将变得非常困难。我们需要清楚地知道，服务出现了什么问题，这些问题出在哪个环节。

为了能解决这些问题，Spring Cloud 提供了 Sleuth 框架作为解决方案，Sleuth 可以与 Zipkin、Apache HTrace 和 ELK 等数据分析、服务跟踪系统进行整合，为服务跟踪、解决问题提供了便利。

10.1.2 服务跟踪系统

目前有许多分布式跟踪系统，例如 Zipkin、HTrace 等，这些系统可以帮助我们收集一些由服务实时产生的数据(主要是日志)，通过这些数据可以分析出分布式系统的健康状态、服务调用过程、调用耗时等指标，为优化系统、解决问题提供了依据。

读者需要区别两个基本的概念：服务跟踪和数据分析，数据分析系统（例如 ELK 等）收集了服务集群所产生的数据后，可以实现服务监控、服务跟踪等功能，但明显数据分析系统的概念更为广泛、抽象。本书将会介绍服务跟踪系统 Zipkin 和著名的数据分析平台 ELK。

10.1.3 Sleuth 的基本概念

Sleuth 借鉴了 Google Dapper 的设计，先来了解以下两个概念。

- Trace：表示整个跟踪过程，从用户发起请求到最终的响应。一次跟踪包含多个跨度，这些跨度以树状结构进行保存。

- Span：跨度，表示一次调用的过程，一次跟踪包含多次调用过程。假设用户向 A 服务发送请求，A 服务又要调用 B 服务，那么此时将会产生两个跨度：用户调用 A 服务、A 服务调用 B 服务。

图 10-1 简单地描述了跨度的概念。

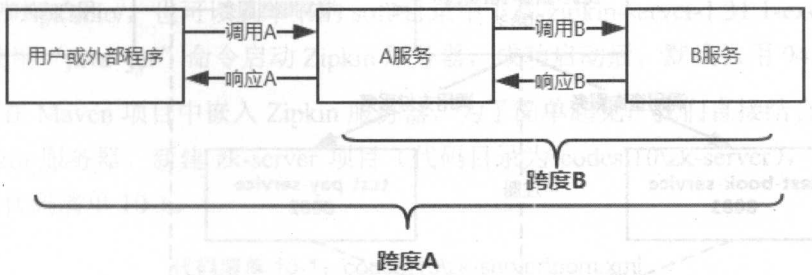


图 10-1 跨度

如图 10-1 所示，用户或外部程序调用 A 服务，此次调用可看作跨度 A，A 服务还要调用 B 服务，在跨度 A 的基础上会产生跨度 B，跨度 B 是跨度 A 的一部分。在 Sleuth 的设计上，跨度 A 是跨度 B 的父跨度。因此在整个跟踪过程中，这些跨度是树状结构的。

除了跟踪和跨度外，还要了解一下 Annotation（事件标识），它主要用于记录事件的存在，主要包括以下几个事件标识。

- cs：Client Sent，表示客户端发送了请求，这个标识意味着跨度的开始。例如前面的 A 服务向 B 服务发送请求，A 服务就是客户端。
- sr：Server Received，表示服务端接收到请求，并开始进行处理。
- ss：Server Sent，表示服务器端完成请求的处理，并对客户端做出响应。
- cr：Client Received，表示客户端接收到响应，意味着整个跨度的结束。

10.1.4 项目准备

在使用 Sleuth 前，先准备本章的测试项目，本章以一个微服务集群为基础，该集群包括以下项目。

- test-eureka-server：Eureka 服务器，端口为 8761。
- test-book-service：图书微服务，主要提供根据 id 查询图书的服务，端口为 8081。
- test-pay-service：支付微服务，主要提供支付服务，端口为 8082。

➢ test-sale-service：销售微服务，会调用图书服务和支付服务，端口为 8083。

以上项目均可以在 codes\10 目录中找到对应的源码，几个项目的结构请见图 10-2。

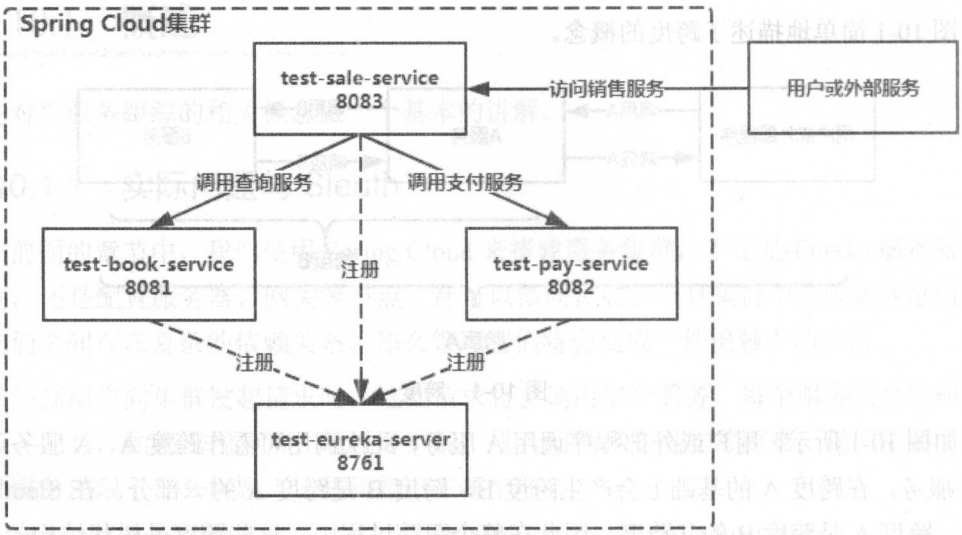


图 10-2　测试项目结构

以上几个测试项目是一个简单的 Spring Cloud 集群，销售服务依赖于“图书服务”和“支付服务”。

10.2　Sleuth 整合 Zipkin

接下来，用 Sleuth 整合 Zipkin，将微服务产生的日志交给 Zipkin 去分析。

10.2.1　Zipkin 简介

Zipkin 是一个分布式跟踪系统，主要用于收集、管理微服务产生的数据。Zipkin 的设计基于 Google Dapper。在实际应用时，我们需要让各个微服务向 Zipkin 服务器报告过程数据。对于 Spring Cloud 来说，已经提供了几个模块来实现数据报告功能，我们仅需要加入依赖，以及做简单配置，即可实现向 Zipkin“写入”数据。

Zipkin 在得到这些数据后，提供了数据查询、分析的功能，这些图形化的功能可以让我们对微服务的调用过程、处理时间、依赖关系等数据一目了然。

10.2.2 构建 Zipkin 服务器项目

启动 Zipkin 服务器，可以选择使用 jar 包的方式，也可以往 Maven 项目中嵌入 Zipkin 服务器。

使用 jar 包启动，需要先下载 Zipkin 的启动 jar 包，读者可以到 Zipkin 的官方网站下载，地址为 http://zipkin.io/，也可以到本书的 soft 目录中下载 zipkin-server-1.31.1-exec.jar。得到 jar 包后，使用“java -jar”命令启动 Zipkin 服务器，成功启动后，默认占用 9411 端口。

也可以在 Maven 项目中嵌入 Zipkin 服务器，为了简单起见，我们直接结合 Sping Boot 来构建 Zipkin 服务器。新建 zk-server 项目（代码目录为 codes\10\zk-server），项目所使用的依赖请见代码清单 10-1。

代码清单 10-1：codes\10\zk-server\pom.xml

```
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-server</artifactId>
</dependency>
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-ui</artifactId>
    <scope>runtime</scope>
</dependency>
```

在 application.xml 中，将启动端口配置为 9411。代码清单 10-2 所示为启动类。

代码清单 10-2：codes\10\zk-server\src\main\java\org\crazyit\cloud\ZkServerApp.java

```
@SpringBootApplication
@EnableZipkinServer
public class ZkServerApp {

    public static void main(String[] args) {
        SpringApplication.run(ZkServerApp.class, args);
    }
}
```

为启动类加上@EnableZipkinServer 注解，运行启动类，访问 http://localhost:9411，可

以看到 Zipkin 的主界面，如图 10-3 所示。

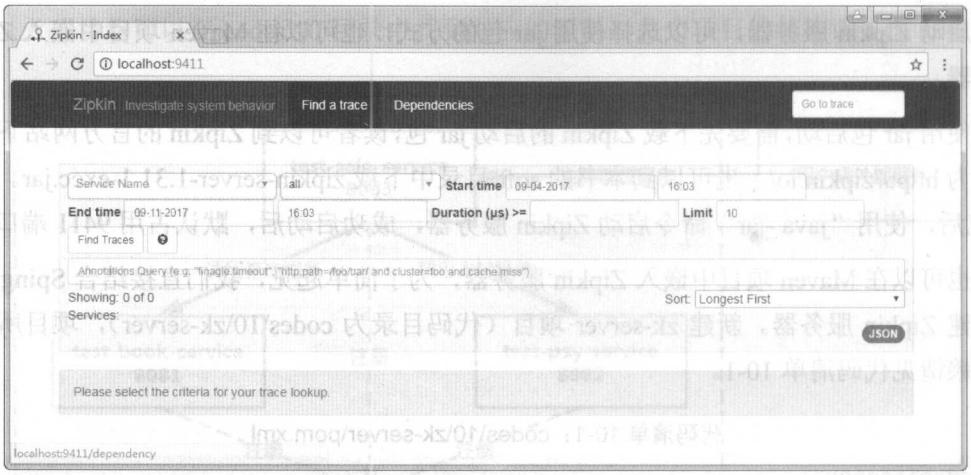


图 10-3　Zipkin 的主界面

10.2.3　配置微服务

接下来需要配置各个微服务，让它们往 Zipkin 服务器写入数据。本章的案例主要有 3 个微服务模块：图书、支付、销售。这几个模块提供了以下服务。

- 图书模块（test-book-service）：提供图书查询服务，地址为/book/{bookId}，返回一个 Book 对象。
- 支付模块（test-pay-service）：提供支付服务，地址为/pay，没有返回，仅做简单的控制台输出。
- 销售模块（test-sale-service）：提供销售服务，地址为/sale/{bookId}，会调用图书模块和支付模块的接口，销售服务的实现请见代码清单 10-3。

代码清单 10-3：codes\10\test-sale-service\src\main\java\org\crazyit\cloud\SaleApplication.java

```
@RequestMapping(method = RequestMethod.GET, value = "/sale/{bookId}")
public String sale(@PathVariable("bookId") Integer bookId) {
    System.out.println("销售模块处理销售");
    // 查找书本
    Book book = bookService.getBook(bookId);
    // 进行支付
    payService.doPay(new BigDecimal(10));
    return "销售成功，书名：" + book.getName() + "，作者：" + book.getAuthor();
}
```

销售模块调用图书、支付模块接口，使用的是 Feign 框架，该框架的使用，读者可参考本书的相关章节，在此不再赘述。

实现了微服务后，在本例的 3 个模块中都加入以下依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

接下来，还要为各个模块配置 Zipkin 服务器，application.yml 的配置如下：

```
spring:
  zipkin:
    base-url: http://localhost:9411
  sleuth:
    sampler:
      percentage: 1.0
```

在 application.yml 中，使用 spring.zipkin.base-url 来配置 Zipkin 的服务器；使用 spring.zipkin.sleuth.sampler.percentage 来配置跨度数据的采样百分比，默认值为 0.1，也就是会向 Zipkin 发送约 10%的跨度数据。本例中为了查看效果，直接将采样百分比配置为 1，也就是全部的跨度数据都会被发送到 Zipkin。在生产环境中，建议按照具体的需求进行抽样，以免增加服务器的负载。

为了能在各个微服务的控制台中看到 Sleuth 的输出，还需要为 3 个微服务配置日志级别，在 application.yml 中加入以下配置：

```
logging:
  level:
    root: INFO
    org.springframework.cloud.sleuth: DEBUG
```

10.2.4 查看数据

配置完 3 个微服务模块后，启动整个集群，在浏览器中访问销售模块 http://localhost:8083/sale/1，可以看到控制台输出相应的字符。以销售模块为例，在访问销售服务后，控制

台会输出以下类似信息：

```
    2017-09-11 16:24:13.251 DEBUG
[sale-service,bd7da949a7aca4c6,2b43e62fc356b5d8,true] 3964 ---
[-book-service-4] .s.c.s.z.ServerPropertiesEndpointLocator : Span will contain
serviceName [sale-service]
    2017-09-11 16:24:13.618 DEBUG
[sale-service,bd7da949a7aca4c6,31c2d63bb091f69e,true] 3964 ---
[x-pay-service-3] .s.c.s.z.ServerPropertiesEndpointLocator : Span will contain
serviceName [sale-service]
    2017-09-11 16:24:13.815 DEBUG
[sale-service,bd7da949a7aca4c6,bd7da949a7aca4c6,true] 3964 ---
[io-8083-exec-10] .s.c.s.z.ServerPropertiesEndpointLocator : Span will contain
serviceName [sale-service]
```

控制台输出的日志是 Sleuth 操作 Span 的日志，例如打开、关闭等。我们调用的销售服务会调用图书和支付模块，因此最终会产生 3 个跨度，也就是产生 3 次调用。访问 Zipkin 的主页，搜索 sale-service 的跟踪记录，可以看到如图 10-4 所示的界面。

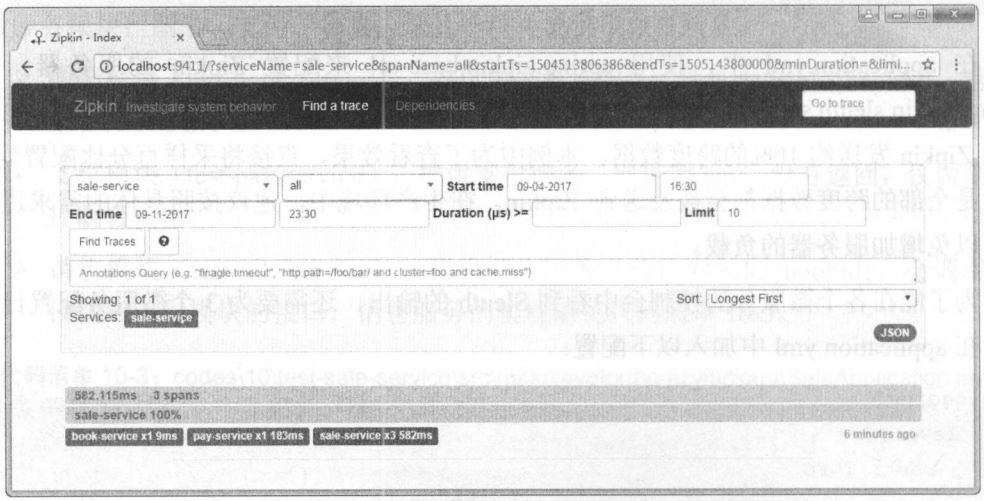


图 10-4 Zipkin 数据

读者在搜索跟踪数据时，注意要选择合适的查找条件。单击图 10-4 所示的跟踪数据，可以看到如图 10-5 所示的界面。

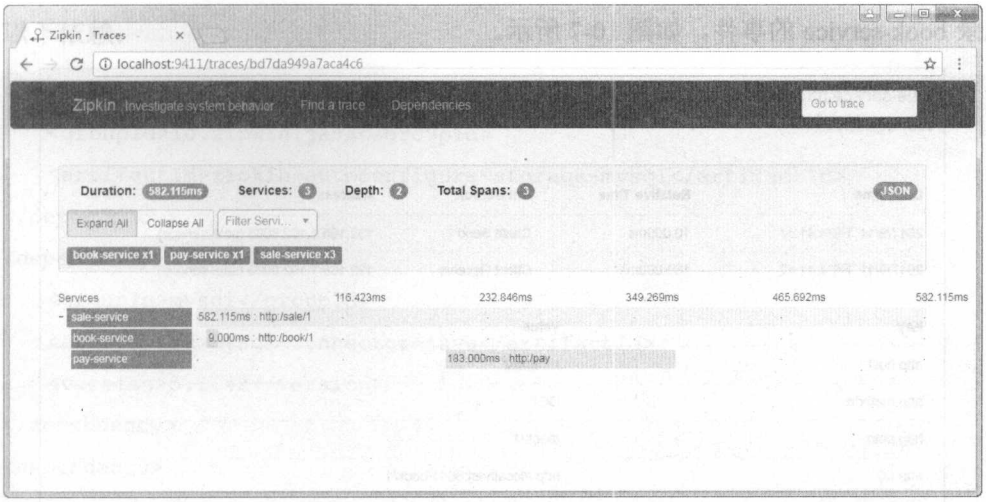


图 10-5 查看跟踪数据

如图 10-5 所示，调用 sale-service 会产生 3 个 Span，界面中显示了调用不同的服务所耗费的时间。选中 book-service 的 Span，查看 Span 的详细信息，如图 10-6 所示。

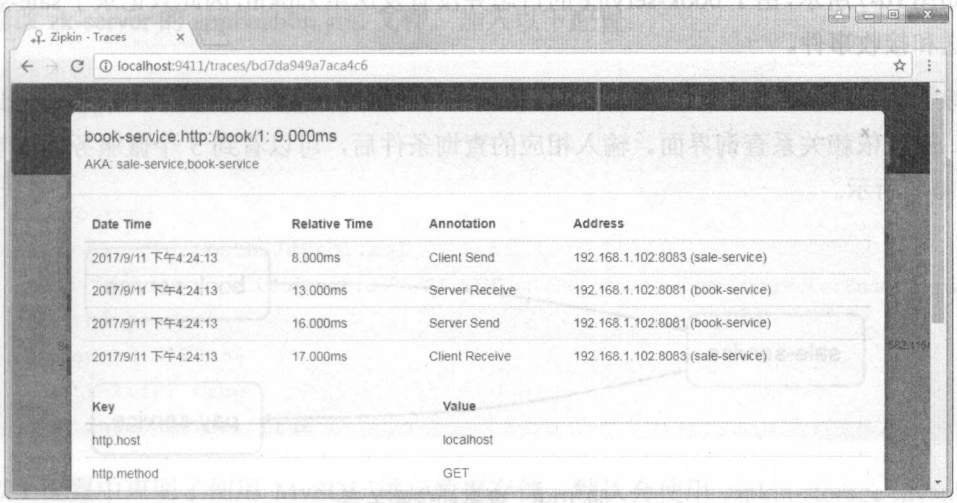


图 10-6 Span 详细信息

根据图 10-6 可知，有 4 个事件标识，客户端（sale-service）发送、服务器（book-service）接收、服务器发送、客户端接收。

在此需要特别说明一下，如果 book-service 中没有加入 Sleuth 或者没有相关配置，那么 Zipkin 将无法收集到该模块的数据，此时 sale-service 向 book-service 发送请求时，将无

法记录 book-service 的事件，如图 10-7 所示。

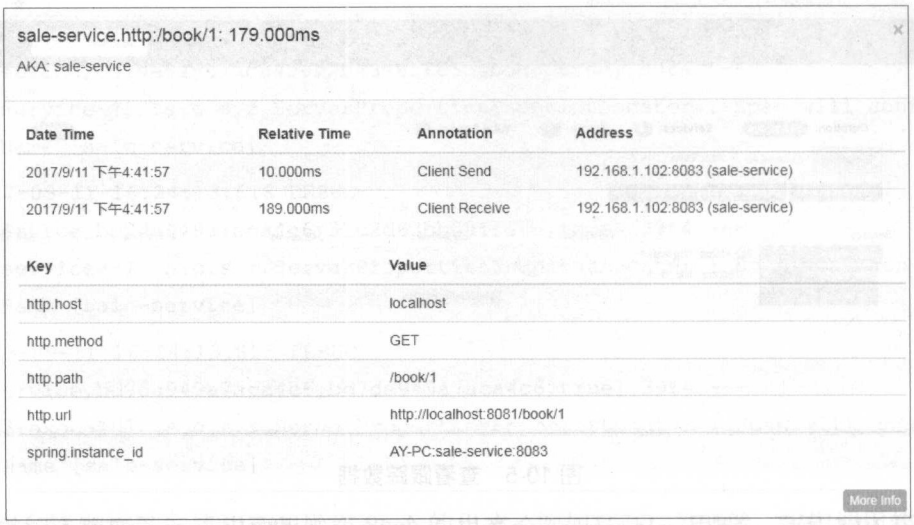


图 10-7　查看 Span 数据

如图 10-7 所示，由于 book-service 的日志并没有发送至 Zipkin，因此只记录了 sale-service 的发送和接收事件。

除了查看跨度数据外，还可以查看服务间的依赖关系。单击主界面上方的 Dependencies 菜单，进入依赖关系查询界面。输入相应的查询条件后，可以看到 3 个微服务的依赖关系如图 10-8 所示。

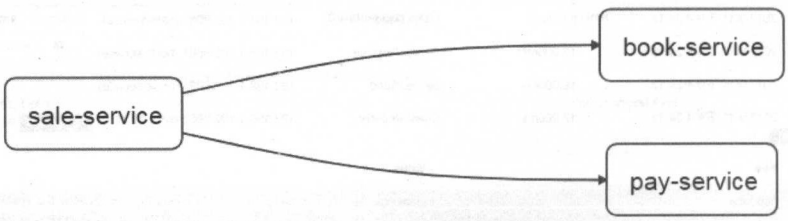


图 10-8　服务的依赖关系

➤➤ 10.2.5　使用 MySQL 保存数据

对于数据量较大的日志，不推荐使用关系型数据库保存，但是，如果仅对部分日志进行采样收集，使用关系型数据库也许更加方便。在默认情况下，Zipkin 会将数据保存到内存中，可以对 Zipkin 进行配置，让其将数据保存至 MySQL 数据库。修改 zk-server 项目，

加入以下依赖：

```
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-storage-mysql</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.42</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
    <version>1.5.4.RELEASE</version>
</dependency>
```

修改 zk-server 的 application.yml 文件，加入以下配置：

```
zipkin:
  storage:
    type: mysql
spring:
  datasource:
    schema: classpath:/mysql.sql
    url: jdbc:mysql://localhost:3306/ZIPKIN?autoReconnect=true&characterEncoding=utf-8
    username: root
    password: 123456
    initialize: true
    continueOnError: true
```

以上配置中声明了使用 MySQL 进行数据存储，默认会使用 zipkin-storage-mysql-*.jar 这个包下面的 mysql.sql 脚本来创建 schema，其他的配置项较为简单。

完成以上工作后，启动 zk-server，在 MySQL 数据库中可以看到已经创建了 Zipkin 的 3 个数据表。再次尝试访问我们的服务，可看到数据表中已经产生相应的数据。

注意：

为了不影响读者试用 zk-server，笔者将相应的依赖、配置注释掉，默认情况下，zk-server 的数据仍然保存在内存中。

10.2.6 使用消息采集数据

细心的读者可能注意到，每一个微服务都需要配置 Zipkin 的地址，服务一多，将会增加运维的工作量。要解决这个问题，可以考虑使用 RabbitMQ 这类消息代理。Spring Cloud 提供了 spring-cloud-sleuth-zipkin-stream 模块，可以由微服务的实例发送消息到 RabbitMQ 中，Zipkin 的服务器作为“消息消费者”来保存这些数据。Zipkin 服务器、微服务实例、RabbitMQ，这 3 者的关系请见图 10-9。

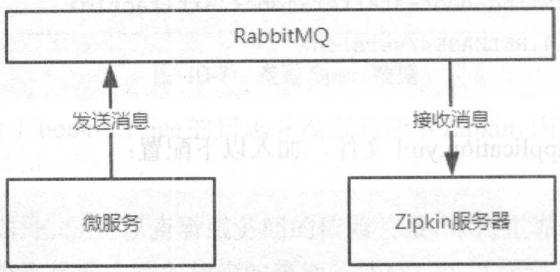


图 10-9 使用 RabbitMQ

为测试消息采集，新建 zk-consumer-server 项目作为 Zipkin 服务器，新建 zk-stream-client 项目作为微服务实例。为微服务实例添加以下依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin-stream</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

修改 zk-stream-client 的 application.yml 文件，内容如下：

```
server:
  port: 8085
spring:
  application:
    name: zk-stream-client
  sleuth:
    sampler:
      percentage: 1.0
```

在配置文件中，只将采样率修改为 1.0，并不需要添加 Zipkin 服务器的配置。默认情况下会连接本地的 RabbitMQ，默认端口为 5762，可以使用以下配置来修改 RabbitMQ 的连接信息：

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
```

接下来，为 Zipkin 的服务器（zk-consumer-server）项目添加依赖：

```
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-server</artifactId>
</dependency>
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-ui</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin-stream</artifactId>
</dependency>
```

将服务器的 Web 端口改为 9411。新建启动类，请见代码清单 10-4。

代码清单 10-4：codes\10\zk-consumer-server\src\main\java\org\crazyit\cloud\ZkStreamServerApp.java

```
@SpringBootApplication
@EnableZipkinStreamServer
public class ZkStreamServerApp {

    public static void main(String[] args) {
        SpringApplication.run(ZkStreamServerApp.class, args);
    }
}
```

在启动类中，使用了@EnableZipkinStreamServer 来替代原来的@EnableZipkinServer 注解。启动 Zipkin 服务器（zk-consumer-server），再启动服务实例（zk-stream-client），访问 http://localhost:8085/hello，再访问 Zipkin 服务器，可以看到已经有相应的 Span 数据生成。

本节对应的案例为 zk-consumer-server 与 zk-stream-client，两个项目都在 codes\10 目录下。

10.3 Sleuth 整合 ELK

本节将讲解如何将微服务的日志传送给 ELK 平台进行分析。

10.3.1 关于 ELK

Elastic 为数据存储、分析提供了一整套解决方案，其中最著名的就是 ELK 系统。ELK 包括 Elasticsearch、Logstash、Kibana 这 3 个项目，这 3 个项目的描述如下。

- Elasticsearch：是一个分布式数据仓库，提供了 RESTful 服务，可用于数据存储、分析。
- Logstash：主要用于数据收集、转换，可将数据保存到指定的数据仓库中。
- Kibana：是可视化的数据管理平台，主要用于操作 Elasticsearch 的数据，它提供了多种图表展示数据，支持动态报表。

微服务所产生的日志数据将会被 Logstash 读取，最终保存到 Elasticsearch 仓库进行保存。ELK 与微服务集群的关系请见图 10-10。

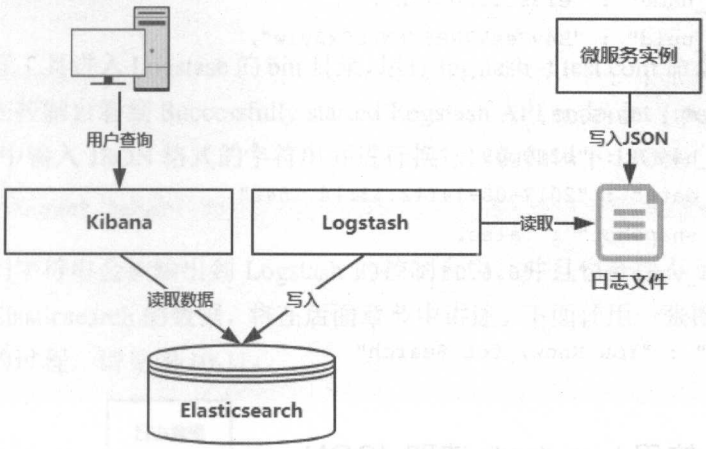


图 10-10 ELK 与微服务集群的关系

接下来，我们将实现图 10-10 所示的整个架构。微服务的集群仍然使用 10.1.4 节的 4 个项目。

10.3.2 下载 ELK

本书所使用的 ELK 各项目的版本均为 5.5.2，读者可以到 Elasticsearch 的官方网站下载，也可以到本书的 soft 目录下载，本书案例所使用的安装包如下：

- elasticsearch-5.5.2.zip
- logstash-5.5.2.zip
- kibana-5.5.2-windows-x86.zip

ELK 的 3 个项目都无须安装，下面讲述 3 个项目的运行。

10.3.3 运行 Elasticsearch

作为数据仓库的 Elasticsearch，是 ELK 的核心，我们先启动 Elasticsearch。本书所使用的是 JDK 1.8，为避免遇到不必要的麻烦，请使用该版本的 JDK。解压 Elasticsearch 的 zip 包，进入 elasticsearch-5.5.2/bin 目录，双击运行 elasticsearch.bat，默认使用 9200 端口。成功启动后，在浏览器中访问 http://localhost:9200，输出如下：

```
{
  "name" : "bVBdBQt",
```

```
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "B4vZeeY7RB6J6WtbSkZylw",
  "version" : {
    "number" : "5.5.2",
    "build_hash" : "b2f0c09",
    "build_date" : "2017-08-14T12:33:14.154Z",
    "build_snapshot" : false,
    "lucene_version" : "6.6.0"
  },
  "tagline" : "You Know, for Search"
}
```

10.3.4 使用 Logstash 读取 JSON

虽然下载后的 Logstash 也不需要安装，但在启动时需要指定数据采集的配置。在 Logstash 的 bin 目录下新建 test.conf 文件，内容如下：

```
input {
    file {
        path => "D:/logs/test.log"
        codec => "json"
    }
}
filter {

}
output {
    stdout {}
    elasticsearch {
        hosts => ["localhost:9200"]
    }
}
```

在 Logstash 中，数据的读取分为 3 个阶段：输入（input）、过滤（filter）和输出（output）。在以上的配置中，表示在 input 阶段会去监听、读取 test.log 文件的 JSON 字符串，将内容进行标准输出（控制台输出）和写入本地的 Elasticsearch 中。本例暂时不对日志数据

进行过滤。

使用命令行工具进入 Logstash 的 bin 目录，运行 logstash -f test.conf 命令启动 Logstash。成功启动后会在控制台看到 Successfully started Logstash API endpoint {:port=>9600}信息。在 test.log 文件中输入 JSON 格式的字符串并进行换行，例如以下 JSON：

```
{"name": "Angus","age": 30}
```

以上输入的字符串会被输出到 Logstash 的控制台中，并且也会保存到 Elasticsearch 仓库，如何查询 Elasticsearch 的数据，将在后面章节中讲述。下面使用一张图来说明 Logstash 进行数据采集的过程，请见图 10-11。

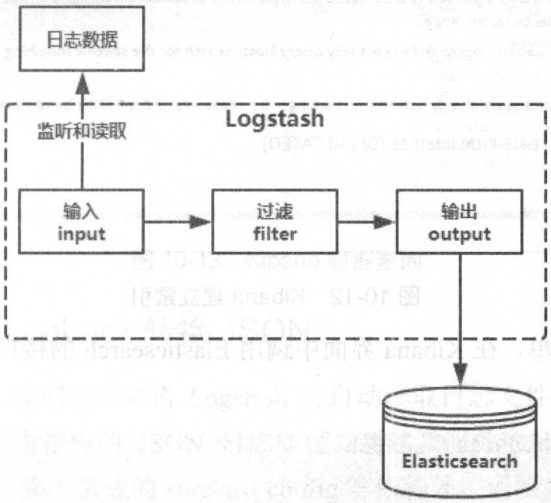


图 10-11 Logstash 采集数据的过程

输入阶段定义“去哪里获取数据”，过滤阶段定义“需要哪些数据”，输出阶段定义“数据保存至何处”，3 个阶段分工明确。微服务所产生的服务跟踪日志将会在 input 阶段进行读取，最后保存到 Elasticsearch 中。

10.3.5 使用 Kibana 展示数据

解压 Kibana 后，进入 bin 目录，运行 kibana.bat 即可启动 Kibana，默认使用 5601 端口。成功启动后，在浏览器中访问 http://localhost:5601，可以看到 Kibana 主界面，按要求建立一个默认的 Index Patterns，即可开始使用 Kibana，如图 10-12 所示。

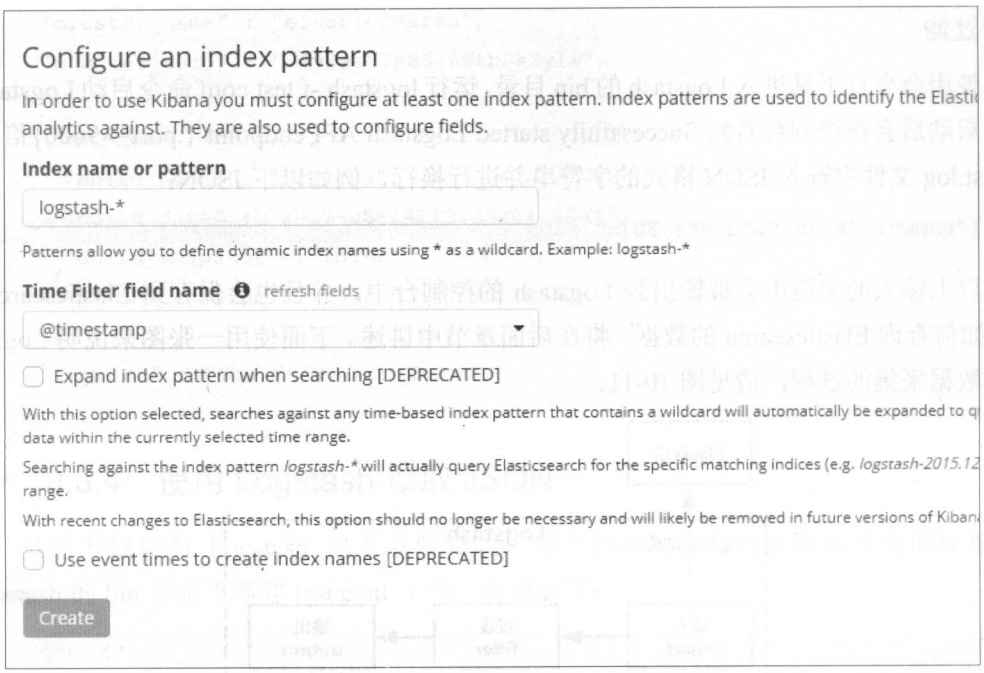


图 10-12 Kibana 建立索引

单击 Dev Tools 菜单，在 Kibana 界面中调用 Elasticsearch 的接口查询数据。在 Console 中输出以下查询语句：

```
GET logstash-2017.09.11/_search
{
  "query": {
    "match": {
      "path": "D:/logs/test.log"
    }
  }
  , "size":30
}
```

运行以上语句，输出如图 10-13 所示。

根据查询结果可知，我们的日志已经被保存到 Elasticsearch。除了以上的数据查询功能外，Kibana 还提供了强大的报表分析功能。可以根据不同的业务，在 Kibana 上定制报表，由于此部分内容并非本书重点，因此不过多讲述。

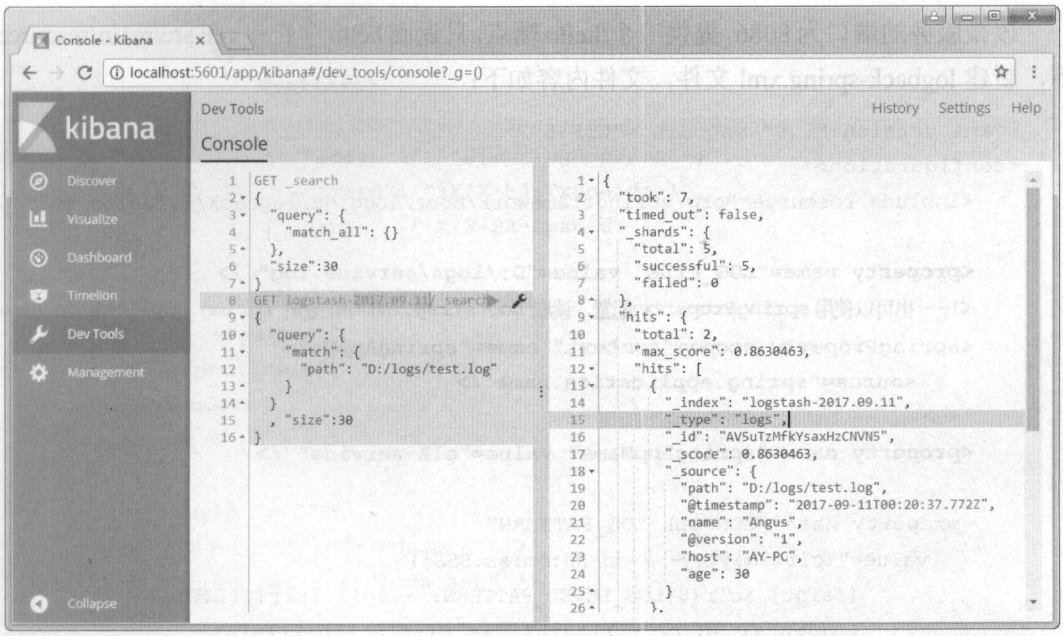


图 10-13 Kibana 数据查询

10.3.6 使用 Logback 转换 JSON

在前面的章节中，我们已经知道 Logstash 会自动读取日志文件中的 JSON。接下来，我们只需要让微服务产生相应的 JSON 日志即可。新建微服务项目 test-elk-service（可在 codes\10 目录下找到），除了常规的 eureka、config 等依赖外，还要为其添加以下依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>4.11</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-core</artifactId>
    <version>1.2.3</version>
</dependency>
```

该微服务的端口为8086，提供一个/hello端点，只返回hello字符串。在src/main/resources下，新建logback-spring.xml文件，文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/defaults.xml" />

    <property name="LOG_FILE" value="D:/logs/service.log" />
    <!-- 也可以使用springProperty配置，读取bootstrap.xml中配置的spring.application.name
    <springProperty scope="context" name="springAppName"
        source="spring.application.name"/>
     -->
    <property name="springAppName" value="elk-service" />

    <property name="CONSOLE_LOG_PATTERN"
        value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS})
            {faint} %clr(${LOG_LEVEL_PATTERN: -%5p}) %clr(${PID:- })
            {magenta} %clr(---){faint} %clr([%15.15t]){faint}
            %clr (%-40.40logger{39}){cyan} %clr(:)
            {faint} %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}" />

    <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
            <level>DEBUG</level>
        </filter>
        <encoder>
            <pattern>${CONSOLE_LOG_PATTERN}</pattern>
            <charset>utf8</charset>
        </encoder>
    </appender>

    <appender name="logstash"
        class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${LOG_FILE}</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${LOG_FILE}.json.%d{yyyy-MM-dd}.zip
            </fileNamePattern>
            <maxHistory>7</maxHistory>
        </rollingPolicy>
        <encoder class="net.logstash.logback.encoder.LogstashEncoder">
```

```
            <providers>
                <pattern>
                    <pattern>
                        {
                        "service": "${springAppName:-}",
                        "trace": "%X{X-B3-TraceId:-}",
                        "span": "%X{X-B3-SpanId:-}"
                        }
                    </pattern>
                </pattern>
            </providers>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="console" />
        <appender-ref ref="logstash" />
    </root>
</configuration>
```

主要注意以上的粗体配置，它可将数据以JSON格式保存到D:/logs/service.log文件中。运行后，会往文件中输出类似如下的日志：

```
{
    "@timestamp": "2017-09-11T22:07:20.647+08:00",
    "@version": 1,
    "message": "Handling span [Trace: e8264176d6068525, Span: e8264176d6068525,
Parent: null, exportable:false]",
    "logger_name":
"org.springframework.cloud.sleuth.instrument.web.TraceHandlerInterceptor",
    "thread_name": "http-nio-8086-exec-1",
    "level": "DEBUG",
    "level_value": 10000,
    "X-Span-Export": "false",
    "X-B3-SpanId": "e8264176d6068525",
    "X-B3-TraceId": "e8264176d6068525"
}
```

由于最终会产生很多日志，但并不是所有的日志都是我们需要的，此时Logstash的filter就派上用场了。我们可以在filter中过滤掉不感兴趣的日志，修改Logstash的配置文件，filter的配置如下所示：

```
filter {
    if ([logger_name] !=
        "org.springframework.cloud.sleuth.instrument.web. TraceHandlerInterceptor") {
        drop {}
    }
}
```

以上的配置表示，logger_name 如果不符合某个字符串，则该日志不会被写入 Elasticsearch 中。运行 Eureka 服务器、再运行 test-elk-service 项目，访问 http://localhost:8086/hello，最终生成的 Span 数据会被写入数据仓库中。数据到达数据仓库后，可以使用 Kibana 来根据具体需求定制报表，此部分内容不在本书讨论范围内，不过多讲述。

10.4 本章小结

随着微服务集群的规模不断增大，服务故障、网络延时等问题也随之增多，如何能够帮助研发、运维人员轻松定位、解决问题，变成一个十分重要的课题。本章以 Spring Cloud Sleuth 框架为基础，讲解了如何整合 Zipkin、ELK 这两个项目。通过本章的学习，可以让读者掌握使用 Zipkin 与 ELK 查看、分析服务所产生的数据，以便更快速地定位、解决问题的方法。

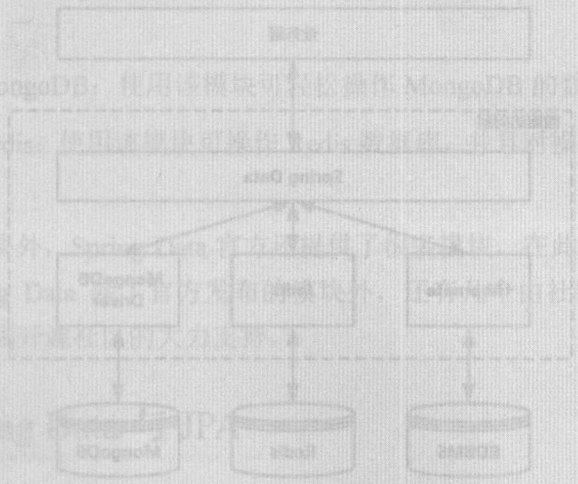
CHAPTER

11

第11章 微服务数据库实战

本章要点

- 介绍 Spring Data
- Spring Data 与 JPA
- Spring Data 与 MongoDB
- Spring Data 与 Redis



实际上到第 10 章为止，Spring Cloud 的内容已经讲述完毕，但是笔者希望本书不仅仅只有 Spring Cloud，更希望能为读者带来一套完整的程序开发方案，例如包括数据库技术、页面层技术等。通过学习本书的知识，希望读者能更快地投入程序开发的工作中。

不论企业应用还是互联网应用，都离不开数据存储，本章将以 Spring Data 项目为基础，讲述在微服务中开发数据库应用。

11.1 概述

我们先对 Spring Data 做一个简单的描述。

11.1.1 关于 Spring Data

Spring Data 框架的目标是为数据的访问提供一个通用的模型。对于程序员来说，不管访问数据库使用的是哪种技术、访问的是哪种数据库，使用了 Spring Data 后，都可以用同样的方式、同样的代码风格来实现对它们的访问，这些数据库包括关系型数据库和非关系型数据库。

读者刚接触 Spring Data 时，可能会觉得 Spring Data 要取代 Hibernate、Jedis 等数据库访问技术。实际上 Spring Data 以这些技术为基础，做进一步的封装，提供类似适配器的功能，让我们可以更加简单地访问数据库，且让代码风格更加统一。Spring Data 与数据库的关系请见图 11-1。

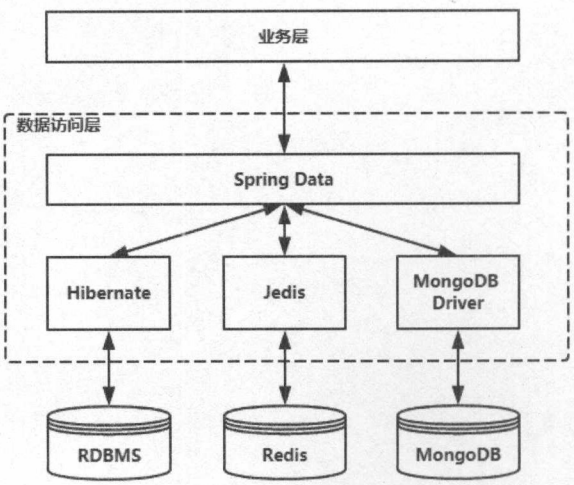


图 11-1 Spring Data

传统 Java EE 的 3 层框架是控制层（Action）、业务层（Service）、数据访问层（DAO）。如图 11-1 所示，Spring Data 的核心作用是为 DAO 层提供统一的数据访问模型。对于业务层的开发人员来说，大多数情况不需要关心底层使用的是哪种数据访问技术。这样的设计，最大的好处是实现了各层的解耦。如果应用程序需要更换数据库，理论上可以一定限度地减少更换的工作量。

11.1.2 Spring Data 的功能

Spring Data 主要有以下几个功能：

- 提供数据与对象映射的抽象层，同一个对象可以映射为不同数据库的数据。
- 根据数据存储接口的方法名，自动实现数据查询。
- 为各个领域模型提供最基本的实现，例如像普通的增、删、改、查功能。
- 可在原有逻辑的基础上，实现自定义的数据库操作逻辑。

11.1.3 Spring Data 的模块

使用 Spring Data 可以实现不同数据库的访问，例如常见的关系型数据库、MongoDB、Redis、Cassandra 等，Spring Data 提供了不同的模块来实现对不同数据库的访问。本章使用 MySQL、MongoDB、Redis 数据库，所涉及的模块如下所述。

- Spring Data JPA：该模块提供了基本的数据操作功能，可减少数据访问层的开发工作量。
- Spring Data MongoDB：使用该模块可轻松操作 MongoDB 的数据。
- Spring Data Redis：使用该模块可操作 Redis 数据库，并且对操作过程进行了极大的简化。

除了以上 3 个模块外，Spring Data 官方还提供了很多模块，在此不一一介绍。作为题外话说明一下，Spring Data 除了官方发布的模块外，还有几个由社区发布的模块，可见 Spring Data 的发展得到开源社区的大力支持。

11.2 Spring Data 与 JPA

JPA 是 Java Persistence API 的简称，是 Sun 在早期推出的持久层规范，目前实现 JPA

规范的主流框架有 Hibernate、OpenJPA 等。Hibernate 框架是当前较为流行的 JPA 实现之一，本节将使用它来访问 MySQL 数据库，使用的 MySQL 版本为 5.6。

11.2.1 构建项目

Spring Boot 提供了一个叫 spring-boot-starter-data-jpa 的模块，我们的项目只要使用这个模块就可以简单地整合 Spring Data JPA、Hibernate 以及其他所需要的模块。新建名称为 spring-jpa 的项目，在 pom.xml 中加入以下依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>1.5.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
    <version>1.5.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.42</version>
</dependency>
```

加入 spring-boot-starter-data-jpa 后，Maven 会自动帮我们引入 Hibernate 5、Spring Data JPA 等包。新建 application.yml 配置文件，配置如下：

```
server:
  port: 8081
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
```

```
    url: jdbc:mysql://localhost/SPRING_DATA
    username: root
    password: 123456
```

本例会到数据库中连接一个名称为 CRA_PERSON 的表，映射的实体为 Person 类。按照传统的 3 层架构新建各个包，请见图 11-2。

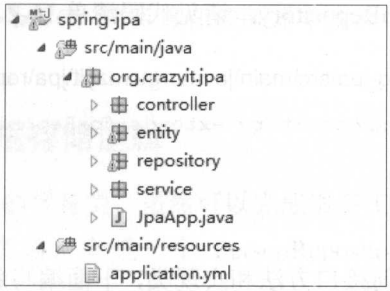


图 11-2 项目结构

如图 11-2 所示，controller 用于存放控制器、entity 用于存放映射实体、service 用于存放业务层对象，其中 repository 用于存放数据库访问对象。

11.2.2 数据访问层与业务层

映射 CRA_PERSON 表的 Person 类，请见代码清单 11-1。

代码清单 11-1：codes\11\spring-jpa\src\main\java\org\crazyit\jpa\entity\Person.java

```
@Entity
@Table(name = "CRA_PERSON")
public class Person {

    @Id
    @GeneratedValue
    private Integer id;

    private String name;

    private Integer age;
```

```
    private String company;
    ……省略 setter 和 getter 方法
}
```

Person 类是一个简单的 Java 对象，使用了 JPA 的相关注解进行修饰。下面在 repository 包中新建数据访问接口 PersonRepository，请见代码清单 11-2。

代码清单 11-2：codes\11\spring-jpa\src\main\java\org\crazyit\jpa\repository\PersonRepository.java

```
public interface PersonRepository extends JpaRepository<Person, Integer> {

}
```

PersonRepository 并无任何接口方法和实现类，下面编写新增和查询的业务方法，请见代码清单 11-3。

代码清单 11-3：codes\11\spring-jpa\src\main\java\org\crazyit\jpa\service\PersonService.java

```
@Service
public class PersonService {

    @Autowired
    PersonRepository personRepository;

    public List<Person> getPersons() {
        List<Person> persons = personRepository.findAll();
        return persons;
    }

    public void save(String name) {
        Person p = new Person();
        p.setName(name);
        p.setAge(33);
        personRepository.save(p);
```

```
    }
}
```

在 PersonService 中注入 PersonRepository，PersonRepository 在继承了 JpaRepository 接口后，就拥有了基本的数据库 CRUD 操作。初次接触的朋友可能觉得比较神奇，实际上 Spring 会为 PersonRepository 接口生成代理类，默认使用 JKD 的动态代理。

控制器默认使用 Spring MVC，可直接在控制器中注入 PersonService 的实例，在此不进行讲述。

11.2.3 自定义数据存储逻辑

前面 Spring 帮我们生成的代理类，虽然可以完成很多工作，但在实际应用时，不可避免要实现自己的数据存储逻辑。新建一个 PersonRepositoryCustom 的接口，让原来的 PersonRepository 接口继承它，在 PersonRepositoryCustom 接口中加入需要自定义的接口方法，请见代码清单 11-4。

代码清单 11-4：

codes\11\spring-jpa\src\main\java\org\crazyit\jpa\repository\PersonRepositoryCustom.java

```
public interface PersonRepositoryCustom {

    List<Person> myQuery();
}
```

修改原来的 PersonRepository 接口，让其同时继承 PersonRepository 和 JpaRepository。新建一个名称为 PersonRepositoryImpl 的实现类，实现 PersonRepositoryCustom 接口，如代码清单 11-5 所示。

代码清单 11-5：

codes\11\spring-jpa\src\main\java\org\crazyit\jpa\repository\impl\PersonRepositoryImpl.java

```
public class PersonRepositoryImpl implements PersonRepositoryCustom {

    @PersistenceContext
    private EntityManager em;
```

```
    public List<Person> myQuery() {
        Query q = em.createQuery("from Person");
        return q.getResultList();
    }
}
```

在实现类中，使用 EntityManager 查询全部的 Person 数据，关于该接口的使用，读者可查阅 JPA 的相关文档。在此要特别说明的是，默认情况下，Spring 会自动帮我们在包中寻找类名为 PersonRepositoryImpl 的实现类，如果使用该类名找不到任何类，则会将接口方法 myQuery 当作 Person 的字段进行数据查询，但 Person 类并没有 myQuery 字段，因此会抛出以下异常：

```
org.springframework.data.mapping.PropertyReferenceException: No property
myQuery found for type Person!
```

在使用时，注意一下命名规则即可。

11.2.4 方法名查询

如果想根据 Person 的某个字符进行查询，实现类似 from Person where name = ?这样的查询，可以直接在 PersonRepository 接口中定义一个 name 方法，请见以下代码：

```
public interface PersonRepository extends JpaRepository<Person, Integer>,
        PersonRepositoryCustom {

    List<Person> name(String name);
}
```

Spring 会自动生成相应的查询方法，我们不需要编写任何实现逻辑。不仅如此，我们还可以在接口的方法名中，通过使用特定的关键字来实现查询功能。例如要查询 name 与 age 等于某值的 Person，可以使用以下的方法名：

```
List<Person> findByNameAndAge(String name, Integer age);
```

又如，要查询 age 小于某值的 Person，可以使用以下方法名：

```
List<Person> findByAgeLessThan(Integer age);
```

Spring 通过定制方法名来实现相关的查询，目前支持二十多个关键字，请见以下表格。

关键字	例子	对应 SQL
And	findByLastnameAndFirstname	where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	where x.lastname = ?1 or x.firstname = ?2
Is,Equals	findByFirstname, findByFirstnameIs, findByFirstnameEquals	where x.firstname = ?1
Between	findByStartDateBetween	where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	where x.age <= ?1
GreaterThan	findByAgeGreaterThan	where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	where x.age >= ?1
After	findByStartDateAfter	where x.startDate > ?1
Before	findByStartDateBefore	where x.startDate < ?1
IsNull	findByAgeIsNull	where x.age is null
IsNotNull,NotNull	findByAge(Is)NotNull	where x.age not null
Like	findByFirstnameLike	where x.firstname like ?1
NotLike	findByFirstnameNotLike	where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	where x.firstname like ?1(parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	where x.firstname like ?1(parameter bound with prepended %)
Containing	findByFirstnameContaining	where x.firstname like ?1(parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	where x.age not in ?1
TRUE	findByActiveTrue()	where x.active = true
FALSE	findByActiveFalse()	where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	where UPPER(x.firstame) = UPPER(?1)

Spring Data 提供的这些特性，使得我们在很多情况下几乎不用编写代码就可以完成查询功能。

11.2.5 使用@Query 注解

如果方法名查询还是无法满足查询要求，可以在方法中使用@Query 注解，提供一段

JPQL（Java Persistence Query Language）语句或 SQL 语句，同样可以实现查询功能，请见以下代码片断：

```
public interface PersonRepository extends JpaRepository<Person, Integer>,
        PersonRepositoryCustom {

    @Query("select p from Person p where p.name = ?1")
    List<Person> findPersonName(String name);
}
```

@Query 注解还支持使用原生的 SQL 查询：

```
@Query(value = "SELECT * FROM CRA_PERSON WHERE NAME = ?1", nativeQuery = true)
List<Person> findNativeByName(String name);
```

以上的接口方法设置了 nativeQuery 属性，可直接使用原生的 SQL 语句进行查询。

除了本节讲述的功能外，Spring Data JPA 模块还提供了很多功能，这些功能的目的都是为了减少数据库开发的工作量，由于篇幅所限，本章不一一列举。

以上的章节，虽然底层使用了 Hibernate 操作 MySQL 数据库，但我们基本上感觉不到 Hibernate 的存在，这也是 Spring Data 所希望达到的效果。接下来，我们将使用同样的模型来操作另外两个著名的数据库。

11.3 Spring Data 与 MongoDB

MongoDB 是一个开源的分布式数据库。经过多年的发展，MongoDB 已经形成稳定的版本，目前，许多著名的公司都选择使用 MongoDB 作为数据存储的解决方案，例如像 Facebook、Bosch 等。

在 11.3.1 节和 11.3.2 节，我们将会讲述如何安装与配置 MongoDB，如果已经有现成 MongoDB 环境的读者，可以跳过这两小节。

11.3.1 安装 MongoDB

本书所使用的 MongoDB 版本为 3.4.9，读者可以到 MongoDB 官网下载，也可以到本书所附的 soft 目录中下载 mongodb-win32-x86_64-2008plus-ssl-3.4.9-signed.msi。

默认会将 MongoDB 安装到 C:\Program Files\MongoDB 目录。使用命令行工具进入 MongoDB\Server\3.4\bin 目录，输入命令 mongod.exe --dbpath=D:\mongo_data\db，其中 --dbpath 用于指定数据目录，看到以下日志，表示已经成功启动：

```
waiting for connections on port 27017
```

默认情况下，MongoDB 并不需要安全认证就可以直接进入数据库，使用下面的步骤创建一个数据库：

- 使用命令行工具，进入 MongoDB\Server\3.4\bin 目录。
- 输入 mongo 命令。
- 使用（创建）数据库命令：user crazyit。
- 写入一条数据：db.runoob.insert({"name":"Angus"})。
- 查看数据库：show dbs。

除了可以使用命令行来管理数据外，还可以使用一些可视化的管理工具，例如可使用 robo。

11.3.2 配置权限

MongoDB 的用户与数据库绑定，可在命令行中建立用户并设置密码。使用 mongo 命令进入 MongoDB 后，输出以下命令新建用户：

```
db.createUser(
  {
    user: "test",
    pwd: "123456",
    roles: [ { role: "readWrite", db: "crazyit" } ]
  }
```

以上命令创建了一个 test 用户，拥有 crazyit 的读写权限。创建用户后，还要开启 MongoDB 的权限认证，修改 MongoDB 服务器的启动命令：

```
mongod.exe --auth --dbpath=D:\mongo_data\db
```

测试用户是否可以登录，使用命令行进入 MongoDB\Server\3.4\bin，输入命令 mongo

localhost/crazyit -utest -p123456，可使用 test 用户成功进入 crazyit 数据库。

11.3.3 MongoDB 的概念

在使用 Spring Data MongoDB 模块时，需要涉及 MongoDB 的一些基本概念，在此先简单了解一下。

- database：数据库，相当于 SQL 的数据库。
- collection：数据集合，相当于 SQL 的数据表。
- document：数据文档，可看作 SQL 的一条数据。
- field：数据域，可看作 SQL 中的 column。
- index：数据索引。

11.3.4 构建项目

与前面的 JPA 类似，Spring Boot 也提供了相应的 Maven 依赖。新建 spring-mongodb 项目，加入以下依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>1.5.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
    <version>1.5.4.RELEASE</version>
</dependency>
```

加入依赖后，Maven 会自动帮我们的项目引入全部依赖，主要的依赖包描述如下。

- spring-data-mongodb：Spring Data 的 MongoDB 模块，版本为 1.10.4。
- mongodb-driver：MongoDB 官方提供的驱动包，版本为 3.4.2。

新建 application.yml 文件，配置如下：

```
server:
  port: 8082
spring:
  data:
    mongodb:
      uri: mongodb://test:123456@localhost:27017/crazyit
```

使用了 uri 属性来连接 MongoDB，格式为 mongodb://user:password@host:port/database。除了 uri 外，还可以使用 host 和 port 进行配置。但要注意的是，MongoDB 的 Java 驱动在 3.0 以后的版本不再支持 spring.data.host 与 spring.data.port 配置，在此笔者建议还是使用 uri 的方式进行配置。

按照之前 JPA 模块的方式，在项目中建立相应的包，用于存放不同的源码：controller、entity、repository、service。

11.3.5 数据访问层与业务层

先在 crazyit 的数据库中新建一个 Person 的 collection，创建命令为 db.createCollection ("Person")。新建 Person 对象，用于映射 collection，请见代码清单 11-6。

代码清单 11-6：codes\11\spring-mongodb\src\main\java\org\crazyit\mongo\entity\Person.java

```
@Document(collection="Person")
public class Person {

    @Id
    private String id;

    private String name;

    private Integer age;

    private String company;
    ……省略 setter 和 getter 方法
}
```

Person 中使用@Document 注解进行修饰，指定了映射 Person 的 collection。Person 实例与前面的 JPA 的类似，只是使用了不同的注解。接下来，编写数据访问接口，请见代码清单 11-7。

代码清单 11-7：codes\11\spring-mongodb\src\main\java\org\crazyit\mongo\repository\PersonRep.java

```
public interface PersonRep extends MongoRepository<Person, String> {


}
```

数据访问接口继承 MongoRepository，无须任何实现，它已经拥有最基本的 CRUD 操作。在业务层中，直接注入 PersonRep 实例，如代码清单 11-8 所示。

代码清单 11-8：codes\11\spring-mongodb\src\main\java\org\crazyit\mongo\service\PersonService.java

```
@Service
public class PersonService {


    @Autowired
    private PersonRep personRep;


    public List<Person> getPersons() {
        List<Person> datas = personRep.findAll();
        return datas;
    }
}
```

除了以上的 findAll 方法外，还有 save、delete 等方法，在此不一一列举。

11.3.6 自定义数据存储逻辑

读者可能已经猜到，如果 Spring 提供的 CRUD 操作无法满足要求，Spring Data MongoDB 模块同样支持自定义数据存储逻辑，就像 JPA 一样。新建自定义的存储接口，请见代码清单 11-9。

代码清单 11-9：

codes\11\spring-mongodb\src\main\java\org\crazyit\mongo\repository\PersonRepCustom.java

```
public interface PersonRepCustom {

    List<Person> myQuery();
}
```

让原来的 PersonRep 接口同时继承 MongoRepository 和 PersonRepCustom，再提供一个实现类，如代码清单 11-10。

代码清单 11-10：

codes\11\spring-mongodb\src\main\java\org\crazyit\mongo\repository\impl\PersonRepImpl.java

```
public class PersonRepImpl implements PersonRepCustom {

    @Autowired
    private MongoTemplate mongoTemplate;

    public List<Person> myQuery() {
        List<Person> datas = mongoTemplate.execute(Person.class,
                new CollectionCallback<List<Person>>() {
                    public List<Person> doInCollection(DBCollection collection)
                            throws MongoException, DataAccessException {
                        // 查询全部数据
                        DBCursor cursor = collection.find();
                        List<Person> result = new ArrayList<Person>();
                        while (cursor.hasNext()) {
                            // 获取源数据实例
                            DBObject source = cursor.next();
                            // 转换为 Person
                            Person p = new Person();
                            ObjectId objectId = (ObjectId)source.get("_id");
                            p.setId(objectId.toHexString());
```

```
                p.setAge((Integer) source.get("age"));
                p.setName((String) source.get("name"));
                p.setCompany((String) source.get("company"));
                result.add(p);
            }
            return result;
        }
    });
    return datas;
  }
}
```

自定义的 myQuery 方法中使用了 mongoTemplate 来执行数据查询。调用 execute 方法时，实现一个 CollectionCallback 接口，这样的使用方式对于接触过 Spring 与 Hibernate 的读者来说非常熟悉。在实现 CollectionCallback 接口时，使用 MongoDB 的 DBCollection 对象进行数据查询操作，本例中直接查询该 Collection 的全部数据，并封装为 Person 集合返回。

11.3.7 方法名查询

JPA 与 MongoDB 模块在数据访问层、自定义数据存储逻辑实现上十分类似，除了这两个功能外，MongoDB 模块也同样支持根据方法名查询数据。在 PersonRep 接口中定义一个查询接口，不需要任何的实现，就可以实现根据 name 字段查询 Person 集合的功能：

```
List<Person> name(String name);
```

同样，也可以使用以下的方法名实现 Less Than 功能：

```
List<Person> findByAgeLessThan(Integer age);
```

除了以上的 LessThan 关键字外，方法名还支持以下关键字。

关键字	例子	对应逻辑
After	findByBirthdateAfter(Date date)	{"birthdate" : {"$gt" : date}}
GreaterThan	findByAgeGreaterThan(int age)	{"age" : {"$gt" : age}}
GreaterThanEqual	findByAgeGreaterThanEqual(int age)	{"age" : {"$gte" : age}}

续表

关键字	例子	对应逻辑
Before	findByBirthdateBefore(Date date)	{"birthdate" : {"$lt" : date}}
LessThan	findByAgeLessThan(int age)	{"age" : {"$lt" : age}}
LessThanEqual	findByAgeLessThanEqual(int age)	{"age" : {"$lte" : age}}
Between	findByAgeBetween(int from, int to)	{"age" : {"$gt" : from, "$lt" : to}}
In	findByAgeIn(Collection ages)	{"age" : {"$in" : [ages…]}}
NotIn	findByAgeNotIn(Collection ages)	{"age" : {"$nin" : [ages…]}}
IsNotNull, NotNull	findByFirstnameNotNull()	{"firstname" : {"$ne" : null}}
IsNull, Null	findByFirstnameNull()	{"firstname" : null}
Like, StartingWith, EndingWith	findByFirstnameLike(String name)	{"firstname" : name} (name as regex)
NotLike, IsNotLike	findByFirstnameNotLike(String name)	{"firstname" : { "$not" : name }} (name as regex)
Containing on String	findByFirstnameContaining(String name)	{"firstname" : name}
NotContaining on String	findByFirstnameNotContaining(String name)	{"firstname" : { "$not" : name}}
Containing on Collection	findByAddressesContaining(Address address)	{"addresses" : { "$in" : address}}
NotContaining on Collection	findByAddressesNotContaining(Address address)	{"addresses" : { "$not" : { "$in" : address}}}
Regex	findByFirstnameRegex(String firstname)	{"firstname" : {"$regex" : firstname }}
Not	findByFirstnameNot(String name)	{"firstname" : {"$ne" : name}}
Near	findByLocationNear(Point point)	{"location" : {"$near" : [x,y]}}
Near	findByLocationNear(Point point, Distance max)	{"location" : {"$near" : [x,y], "$maxDistance" : max}}
Near	findByLocationNear(Point point, Distance min, Distance max)	{"location" : {"$near" : [x,y], "$minDistance" : min, "$maxDistance" : max}}
Within	findByLocationWithin(Circle circle)	{"location" : {"$geoWithin" : {"$center" : [[x, y], distance]}}}
Within	findByLocationWithin(Box box)	{"location" : {"$geoWithin" : {"$box" : [[x1, y1], x2, y2]}}}
IsTrue, True	findByActiveIsTrue()	{"active" : true}
IsFalse, False	findByActiveIsFalse()	{"active" : false}
Exists	findByLocationExists(boolean exists)	{"location": {"$exists" : exists }}

以上表格中的“$gt”“$gte”“$lt”“$lte”为 MongoDB 的查询操作符，关于这些操作符的作用，在此不展开讲解，读者可自行查阅 MongoDB 的文档。

在数据访问层接口的方法名中使用这些关键字，可以帮我们实现各种查询功能，一方

面减轻了编码压力，另一方面也减少了出错的几率。

11.3.8 使用@Query注解

定义接口无法满足查询需求，可以使用@Query注解来定义查询条件，请见以下代码片断：

```
@Query("{ 'name' : ?0, 'age' : ?1 }")
List<Person> findByNameAndAge(String name, Integer age);
```

为@Query注解传入查询的JSON语句即可实现，除了查询外，还可以定制查询的字段，如以下代码片断：

```
@Query(value = "{ 'name' : ?0 }", fields = "{ 'name' : 1, 'company' : 1}")
List<Person> findByName(String name);
```

以上的@Query注解添加了fields字段，表示最终查询出来的Person只有id、name、company属性，其他的属性值为null。

注意：

虽然都是@Query注解，而且使用方式类似，但是与JPA的@Query注解是两个完全不同的注解，读者在使用时请勿混淆。

Spring Data MongoDB模块还有很多强大的功能，限于篇幅，本章不一一列举。接下来，讲述Spring Data与Redis。

11.4 Spring Data与Redis

本节介绍使用Spring Data Redis模块。Redis是一款开源的key-value数据库，除了key-value外，还支持例如set、hash等数据结构。目前有许多Redis客户端可供选择，在Java领域，使用较为广泛的客户端有Redisson、Jedis，Spring Data Redis模块默认使用Jedis。

11.4.1 Redis的安装与配置

Redis的正式版不支持Windows，但是在Git上面可以找到一个Windows版本的Redis服务器（只支持64位的Windows），本节将使用其作为Redis服务器，读者可以在本书的

soft 目录下载 Redis-x64-3.2.100.zip 压缩包。

将下载的 Redis 服务器压缩包解压到具体目录，使用命令行工具进入目录，运行 redis-server.exe redis.windows.conf，Redis 服务器启动后如图 11-3 所示。

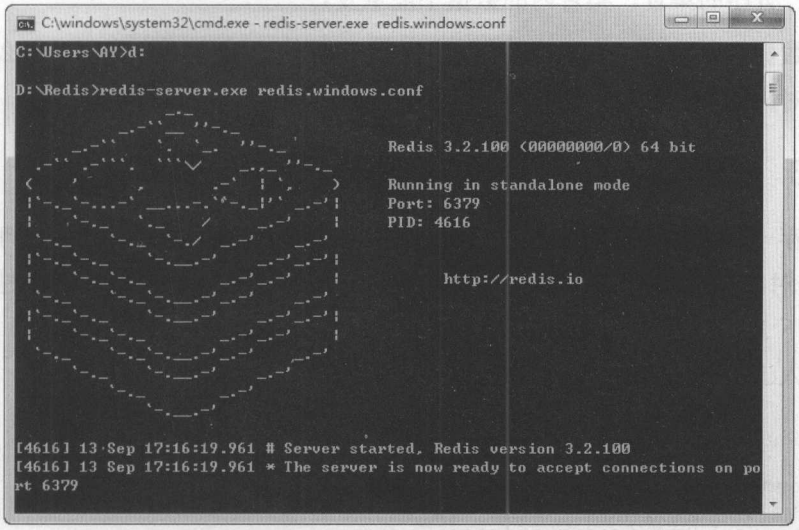


图 11-3 启动 Redis

如图 11-3 所示，默认占用 6379 端口。默认情况不需要输入密码即可进入 Redis，可以修改配置文件 redis.windows.conf，修改或添加属性 requirepass 123456。如果需要修改连接端口，也可以修改该配置文件的 port 属性，修改完配置后需要重启 Redis 服务器。

启动 Redis 服务器后，使用命令行工具进入 Redis 目录，执行 redis-cli.exe –a 123456，即可登录到 Redis。在 Redis 客户端的命令行中，输入 set name_a test_a，表示往 Redis 写一条 key 为 name_a 的 string 类型数据，再使用 get name_a 可以获取该数据的值。

目前 Redis 有许多图形化的管理工具，读者可以选择适合自己的，笔者在此推荐使用 RedisStudio，读者可以自行下载使用。

11.4.2 Redis 的数据类型

Redis 使用 key-value 的结构保存数据，其中 value 支持 5 种数据类型：string、hash、list、set、zset。下面对这 5 种数据类型做一个简单的描述。

- string：最基本的数据类型，可保存任何数据。
- hash：一个键值对的集合，集合中以字段名作为 key，字段值作为 value，主要用于保存对象。

- list：字符串列表，可以往列表中添加元素。
- set：无序的集合，集合内的数据不能重复。
- zset：有序的集合，集合内的数据不能重复。

在进行数据存储时要注意数据类型，例如保存时使用的是 hash 类型，那么在获取数据时也要使用对应的类型。

11.4.3 使用 Jedis

如果保存一些不会重复的数据（例如 UUID），使用 set 类型来保存较为理想；如果保存的是 Java 对象，则使用 hash 类型较为合适。Spring Data 默认使用 Jedis 来操作 Redis，我们先看一下 Jedis 如何操作 hash 与 set 类型的数据。新建一个名为 test-jedis 的 Maven 项目，为其加入 Jedis 的依赖：

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.9.0</version>
</dependency>
```

分别读写 hash 与 set 类型的数据，请见代码清单 11-11。

代码清单 11-11：codes\11\test-jedis\src\main\java\org\crazyit\cloud\JetisTest.java

```
public class JetisTest {

    public static void main(String[] args) {
        // 登录本地的 Redis
        Jedis jedis = new Jedis("localhost");
        jedis.auth("123456");
        // 将数据保存为 hash 类型
        Map<String, String> data = new HashMap<String, String>();
        data.put("name", "Angus");
        data.put("age", "33");
        data.put("company", "crazyit");
```

```
        jedis.hmset("person_test", data);                                    ①
        // 查询 hash 类型的数据
        List<String> dbDatas = jedis.hmget("person_test", "name", "age");    ②
        for(String dbData : dbDatas) {
            System.out.println(dbData);
        }
        /* =========分隔线==========  */
        // 将数据保存为 set 类型
        jedis.sadd("person_test_ids", "1", "2");                             ③
        // 查询数据
        Set<String> dbDatasSet = jedis.smembers("person_test_ids");          ④
        for(String dbData : dbDatasSet) {
            System.out.println(dbData);
        }
    }
}
```

代码清单中的①和②，分别为写入和读取 hash 类型的数据，③和④分别为写入和读取 set 类型的数据。运行代码清单 11-11 后，使用 RedisStudio 工具分别查看添加的数据，如图 11-4 与图 11-5 所示。

Key: person_test
Type: hash
Size: 3
TTL: -1

Field	Value
company	crazyit
age	33
name	Angus

图 11-4 hash 数据

Key:	person_test_ids
Type:	set
Size:	2
TTL:	-1
	Value
	1
	2

图 11-5 set 数据

11.4.4 构建 Spring Data 项目

Spring Boot 同样提供了 Spring Data 与 Redis 的依赖，新建名称为 spring-redis（可在 codes/11 目录下找到）的 Maven 项目，加入以下依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>1.5.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <version>1.5.4.RELEASE</version>
</dependency>
```

新建 application.yml 文件，内容如下：

```
server:
  port: 8083
spring:
  redis:
    port: 6379
    password: 123456
    host: localhost
```

使用了 spring.redis.x 来配置 Redis 的相关属性，除了以上配置外，还可以使用 spring.redis.url 属性，格式为 redis://user:password@host:port，其中 user 随便填写，会被忽略。上面的配置等价于以下配置：

```
url: redis://ignore:123456@localhost:6379
```

与前面的 JPA、MongoDB 一样，为项目建立相应的包：controller、repository、service、entity。

11.4.5 数据访问层与业务层

新建一个 Person 类，用于映射 Redis 的数据，请见代码清单 11-12。

代码清单 11-12：codes\11\spring-redis\src\main\java\org\crazyit\redis\entity\Person.java

```
@RedisHash("Person")
public class Person {

    @Id
    private String id;

    @Indexed
    private String name;

    private Integer age;

    private String company;
    ……省略 setter 和 getter 方法
}
```

Person 类中使用了@RedisHash 注解，表示 Person 对象将会以 hash 类型保存。实际上，在实现时，Person 数据的 id 将会作为 set 类型进行保存。还需要注意 Person 类的 name 属性，使用了@Indexed 注解，表示在保存 Person 时，name 属性将会建立索引。

接下来，编写数据访问接口，请见代码清单 11-13。

代码清单 11-13：codes\11\spring-redis\src\main\java\org\crazyit\redis\repository\PersonRep.java

```
public interface PersonRep extends CrudRepository<Person, String> {


}
```

数据访问接口继承了CrudRepository，同样不需要任何的实现，PersonRep接口即拥有普通的CRUD操作。编写业务类，添加保存与查询方法，请见代码清单11-14。

代码清单11-14：codes\11\spring-redis\src\main\java\org\crazyit\redis\service\PersonService.java

```
@Service
public class PersonService {

    @Autowired
    private PersonRep personRep;

    public List<Person> getPersons() {
        Iterable<Person> persons = personRep.findAll();
        List<Person> datas = new ArrayList<Person>();
        for(Iterator<Person> it = persons.iterator(); it.hasNext();) {
            Person p = it.next();
            datas.add(p);
        }
        return datas;
    }

    public void save(String name) {
        Person p = new Person();
        p.setName(name);
        p.setAge(33);
        p.setCompany("crazyit");
        personRep.save(p);
    }
}
```

代码清单11-14中的粗体代码调用了查找和保存方法。注意一下，CrudRepository接口的findAll方法返回的是Iterable实例，需要对其遍历并将Person放到List中。为了能够查看数据保存到Redis的类型，可调用save方法来保存Person。

保存两次 Person 到 Redis，使用 RedisStudio 查看，可以看到数据如图 11-6 和图 11-7 所示。

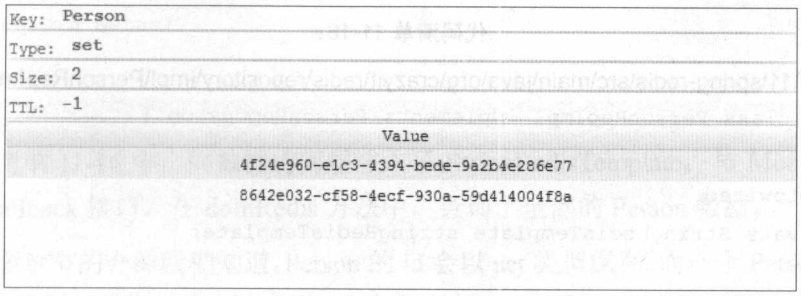


图 11-6 保存的 id

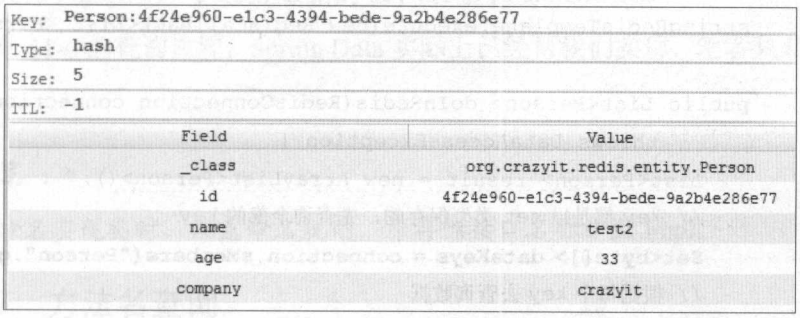


图 11-7 保存的数据

根据数据保存的结构可知，所有 Person 的 id 使用 set 类型进行保存，而具体的 Person 数据则会使用 hash 类型进行保存。一个 Person 实例的 key 格式为：Person:UUID。读者看到这里可能已经知道，笔者为什么在 11.4.3 节特别讲述 Redis 的 set 与 hash 类型数据。

11.4.6 自定义数据存储逻辑

新建一个 PersonRepoCustom 接口，用于定义自定义的数据存储接口，请见代码清单 11-15。

代码清单 11-15：codes\11\spring-redis\src\main\java\org\crazyit\redis\repository\PersonRepCustom.java

```
public interface PersonRepCustom {


    List<Person> myQuery();
}
```

将原来的 PersonRep 接口修改为同时继承 CrudRepository 和 PersonRepCustom，新建 PersonRepCustom 接口的实现类，如代码清单 11-16。

代码清单 11-16：

codes\11\spring-redis\src\main\java\org\crazyit\redis\repository\impl\PersonRepImpl.java

```
public class PersonRepImpl implements PersonRepCustom {

    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    public List<Person> myQuery() {
        List<Person> datas =
            stringRedisTemplate.execute(new RedisCallback<List <Person>>() {

            public List<Person> doInRedis(RedisConnection connection)
                    throws DataAccessException {
                List<Person> result = new ArrayList<Person>();
                // key 都是以 set 类型保存的，先查询全部的 key
                Set<byte[]> dataKeys = connection.sMembers("Person".getBytes());
                // 根据每个 key 去查询数据
                for(byte[] dataKey : dataKeys) {
                    // 数据是以 hash 类型保存的，key 使用以下格式："Person:UUID"
                    String dataKeyStr = "Person:" + new String(dataKey);
                    Map<byte[], byte[]> data =
                        connection.hGetAll(dataKeyStr. getBytes());
                    // 读取数据，并转换为 Person
                    String age = new String(data.get("age".getBytes()));
                    String name = new String(data.get("name".getBytes()));
                    String company = new String(data.get("company".getBytes()));
                    String id = new String(data.get("id".getBytes()));
                    Person p = new Person();
                    p.setId(id);
                    p.setAge(Integer.parseInt(age));
                    p.setName(name);
                    p.setCompany(company);
                    result.add(p);
                }
```

```
                return result;
            }
        });
        return datas;
    }
}
```

在代码清单 11-16 中，向数据访问层注入了 StringRedisTemplate，与 MongoDB 类似，实现 RedisCallback 接口，在 doInRedis 方法中，查询了全部的 Person 数据。

根据前面章节的介绍我们知道，Person 的 id 会以 set 类型保存，而一个 Person 会以 hash 类型保存，因此在实现具体的查询逻辑时，先查询全部 id，再根据 id 去查询每一个 Person。注意在查询具体 Person 时，Person 数据的 key 格式为 Person:UUID。

代码清单 11-6 的查询过程，Spring Data 实际上已经帮我们实现，笔者只是将该过程重新展现而已。

在命名实现类时，注意命名规则，原存储接口名称加上 Impl。

11.4.7 方法名查询

Redis 模块同样支持方法名查询，例如要根据 Person 的 name 来查询，可定义以下接口方法：

```
List<Person> findByName(String name);
```

使用这种方式进行查询，name 字段必须要建立索引。

在方法名中还可以使用以下关键字。

- And：例如在接口中可以定义 findByNameAndAge。
- Or：例如 findByNameOrAge。
- Is、Equals：例如 findByNameIs、findByName、findByNameEquals。
- Top、First：例如 findFirst5Name、findTop5ByName，实现查询前 5 条记录。

除了本章所介绍的功能外，Spring Data Redis 模块还有很多其他功能，由于篇幅所限，不再列举。

11.5 本章小结

本章主要以 Spring Data 为基础，讲述了如何在 Spring Boot 中开发数据库应用。本章主要介绍了如何使用 Spring Data 操作 MySQL、MongoDB 和 Redis。根据本章的例子可知，Spring Data 提供了一种统一的编写风格，让我们几乎可以使用同样的模型来开发数据库应用。

本章的案例主要基于 Spring Boot，因此可以很轻松地将本章的案例配置为微服务进行部署。学习完本章后，再结合前面的 Spring Cloud 知识，相信读者已经可以使用 Spring Cloud 加 Spring Data 来开发各种企业应用或者互联网应用程序。

CHAPTER

12

第12章 案例实战

本章要点

- 案例架构简述
- 案例技术简介
- 在 Spring Boot 中使用 JSP
- Spring Boot 与 Thymeleaf
- 图书管理案例

前面章节讲解了微服务开发、数据库应用，本章前面将会讲解表现层技术，然后会以一个小案例来结束本书的讲解。

12.1 概述

首先对本章的技术以及案例做一个简单的描述。

12.1.1 表现层技术

Spring Boot 支持多种表现层技术，例如像 FreeMarker、Thymeleaf、JSP 等。JSP 作为传统的表现层技术，已经被广泛应用到各类系统中，很多公司已经形成以 JSP 为基础的开发框架。虽然 JSP 是官方标准，但是掩盖不了 JSP 的一些缺点：网页资源与前端程序无法分离、过度依赖 Web 容器等。目前有许多模板技术可以取代 JSP，例如 FreeMarker、Velocity。在本章的 12.2 节，将会讲述 Spring Boot 与 JSP 的整合使用，在 12.3 节，会讲述 Spring 官方推崇的 Thymeleaf 框架。

12.1.2 案例概述

在本章的最后一节，我们会在一个简单的图书管理案例中使用 Spring Cloud、Spring Data、Thymeleaf 等技术。案例涉及功能不多，但笔者希望能为大家提供最基础的开发框架。本章案例主要有以下功能：

- 前台图书展示。
- 后台图书列表。
- 后台添加图书功能。
- 后台用户登录功能。

12.1.3 案例技术选型

在本章最后会开发一个完整的案例，包括表现层、业务层、数据库等，案例使用以下技术：

- Thymeleaf 作为模板引擎。
- Spring MVC。
- Spring Cloud 的主要框架。

- Spring Data。
- MySQL 数据库。

下面，先介绍两个重要的表现层技术：JSP 与 Thymeleaf。

12.2 Spring Boot 与 JSP

Spring Boot 的官方文档并不是很推荐使用 JSP 作为表现层，但仍然提供了 JSP 的开发示例，本节将讲述在 Spring Boot 中使用 JSP，如已掌握该部分知识，可跳过本节。

12.2.1 构建项目

新建一个 Maven 项目，pom.xml 的内容如代码清单 12-1 所示。

代码清单 12-1：codes\12\spring-boot-jsp\pom.xml

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.4.RELEASE</version>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-jasper</artifactId>
    </dependency>
</dependencies>
```

12.2.2 配置

为项目增加视图配置，请见代码清单 12-2。

代码清单 12-2：codes\12\spring-boot-jsp\src\main\resources\application.yml

```
spring:
  mvc:
    view:
      prefix: /pages/
      suffix: .jsp
```

Web 项目的结构如图 12-1 所示。

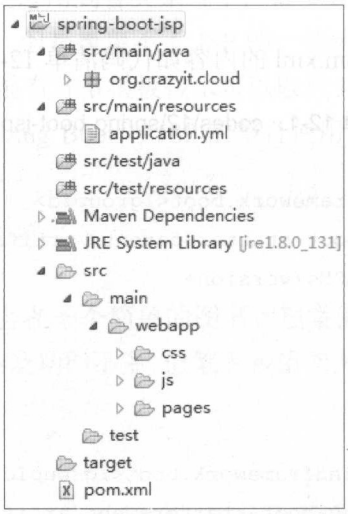


图 12-1 项目结构

根据图 12-1 可知，spring-boot-jsp 是一个常规的 Web 项目，只是与 Spring Boot 整合后，在开发环境中可以直接运行启动类的 main 方法，即可启动 Web 容器予以调试。使用 Spring Boot 将应用打成 war 包，还需要让启动类继承 SpringBootServletInitializer，本例的启动类请见代码清单 12-3。

代码清单 12-3：codes\12\spring-boot-jsp\src\main\java\org\crazyit\cloud\ServerApp.java

```
@SpringBootApplication
@RestController
public class ServerApp extends SpringBootServletInitializer {
```

```
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application;
    }

    public static void main(String[] args) {
        new SpringApplicationBuilder(ServerApp.class).run(args);
    }
}
```

如果启动类不继承 SpringBootServletInitializer，那么打出来的 war 包，在 Tomcat 下将无法访问。

12.2.3 打包部署

本书前面的案例都是在项目中内嵌 Web 容器（Tomcat 或 Jetty 均可），最终会把项目打成一个可执行的 jar 包。在生产环境中，可以使用“java –jar”命令来运行。Spring Boot 不支持（不建议）让 JSP 运行在内嵌的 Web 服务器中，如果 Spring Boot 整合了 JSP，则需要将项目打成一个 war 包，并将其放到 Tomcat 中运行。对于使用了 JSP 的 Spring Boot 项目，结构请见图 12-2。

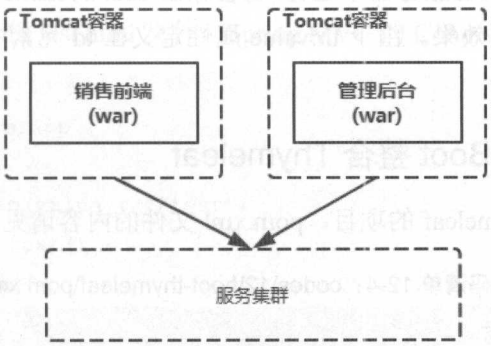


图 12-2 war 包与服务集群

如图 12-2 所示，表现层项目将会打成一个 war 包部署到 Tomcat 容器中。这样做对于整个项目而言，最终会形成有些模块打成可执行的 jar 包，有些模块则打成 war 包进行另外的部署，这会增加维护的工作量。

如果使用 FreeMarker、Thymeleaf 等模板引擎，则可以将表现层项目打成一个普通的 jar 包，jar 包中可以内嵌 Web 容器，这样就不需要再进行额外的部署工作。接下来，将讲述

Spring Boot 推荐使用的模板引擎 Thymeleaf。

12.3 模板引擎 Thymeleaf

本节将介绍 Thymeleaf 的基本使用方法。

12.3.1 关于 Thymeleaf

Thymeleaf 是一个新颖的模板引擎，它提供了一套很接近 HTML 的标签属性。使用这些属性，可以极大地减少程序对界面的侵入，程序员与网页设计师（美工）能更好地协作，加入了程序逻辑的页面，美工也可以直接在浏览器中打开查看。以下是一段使用了 Thymeleaf 属性的 HTML 代码：

```
<table>
    <tr>
        <td th:value="${data.name}">小明</td>
    </tr>
</table>
```

以上的 HTML 代码放在服务器中运行，将会输出 data 的 name 属性。在浏览器中打开，则可以查看到“小明”的效果。由于 th:value 属性定义在 td 元素中，因此在界面上，对展示效果没有丝毫影响。

12.3.2 Spring Boot 整合 Thymeleaf

新建名称为 boot-thymeleaf 的项目，pom.xml 文件的内容请见代码清单 12-4。

代码清单 12-4：codes\12\boot-thymeleaf\pom.xml

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.4.RELEASE</version>
</parent>
<properties>
    <thymeleaf.version>3.0.2.RELEASE</thymeleaf.version>
    <thymeleaf-layout-dialect.version>2.1.1</thymeleaf-layout-dialect.version>
</properties>
```

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
</dependencies>
```

需要注意的是，spring-boot-starter-thymeleaf默认使用的是Thymeleaf 2.1。为了能使用新版本（3.0）的Thymeleaf，需要覆盖父项目的两个属性，如代码清单12-4中的粗体配置。为项目加入Thymeleaf的配置，请见代码清单12-5。

代码清单12-5：codes\12\boot-thymeleaf\src\main\resources\application.yml

```
spring:
  thymeleaf:
    cache: false
    mode: HTML
```

代码清单12-5所示的两个配置，cache表示是否开启模板缓存，如果在开发环境，一般设置为false，mode则用于设置模板的模式。

在Spring Boot中使用模板引擎，默认要将模板放到src/main/resources/templates目录中。例如在templates下有一份test.html文件，Spring MVC的Controller实现如下：

```
@Controller
public class ServerApp {

    @RequestMapping(value = "/test")
    public String test() {
        return "test";
    }
}
```

如果想修改模板存放的目录，可以使用以下配置：

```
spring.thymeleaf.prefix=classpath:/myTtemplates/
```

12.3.3 加载资源

项目中会存在一些CSS、JS、图片等静态资源，可以使用spring.resources.static-locations来配置存放地址，以下为该属性的默认值：

```
    spring.resources.static-locations=classpath:/META-INF/resources/,classpath:/
resources/,classpath:/static/,classpath:/public/
```

换言之，像 CSS 这类静态资源，可以放到 src/resources/static 等目录。在 HTML 模板中，可以使用以下的 Thymeleaf 属性来指定资源：

```
    <link rel="stylesheet" th:href="@{/css/test.css}" href="../css/test.css" type=
        "text/css" />
    <script type="text/javascript" th:src="@{/js/test.js}" src="../js/test.js">
        </script>
```

12.3.4 获取请求数据

在 Controller 中，经常会将数据设置到 HttpServletRequest、HttpSession、ServletContext 中，例如 Controller 会有以下代码：

```
    @RequestMapping(value = "/value")
    public String setValue(HttpServletRequest request, HttpSession session) {
        request.setAttribute("valueA", "testA");
        session.setAttribute("valueB", "testB");
        session.getServletContext().setAttribute("valueC", "valueC");
        return "value";
    }
```

在模板中使用以下方式获取以上的设置的数据：

```
    <span th:text="${valueA}"></span></br>
    <span th:text="${session.valueB}"></span></br>
    <span th:text="${#servletContext.getAttribute('valueC')}"></span>
```

12.3.5 调用 Bean 方法

Thymeleaf 可以直接调用 Spring 的 Bean 方法，例如有以下的 Bean：

```
    @Service(value = "myService")
    public class MyService {

        public User getUser(String name) {
            User u = new User();
            u.setUserName(name);
            u.setPassword("password");
```

```
        return u;
    }
}
```

在模板中，可以使用以下方式调用以上的 getUser 方法：

```
<span th:text="${@myService.getUser('crazyit').userName}"></span>
```

12.3.6 遍历集合

在模板中遍历数据集合也非常方便。例如在 Controller 中设置了一个 User 集合，在模板中可使用以下方式遍历集合并获取相应的属性值：

```
<table style="text-align: center;" border="1">
    <tr>
        <th>名称</th>
        <th>密码</th>
    </tr>
    <tr th:each="user : ${users}">
        <td th:text="${user.userName}"></td>
        <td th:text="${user.password}"></td>
    </tr>
</table>
```

直接在需要遍历的 HTML 元素中加入 th:each 属性，在需要显示的 HTML 元素中加上 th:text 属性。

12.3.7 表单提交

提交表单是最常用的功能，在模板中使用以下代码来定义一个表单：

```
<form action="#" th:action="@{/login}" th:object="${loginUser}" method="post">
    <input type="text" name="userName" th:value="${loginUser?.userName}" /></br>
    <input type="text" name="password" th:value="${loginUser?.password}" /></br>
    <input type="submit" value="登录"/>
</form>
```

接收请求的 Controller 方法实现如下：

```
@RequestMapping(value = "/login")
public String login(@ModelAttribute User loginUser, Model model,
```

```
        HttpSession session) {
    if(loginUser.getUserName() == null || "".equals(loginUser.getUserName())) {
        model.addAttribute("message", "fail");
        // 失败，将原信息返回
        model.addAttribute("loginUser", loginUser);
    } else {
        System.out.println(loginUser.getUserName() + "---" + loginUser.getPassword());
        model.addAttribute("message", "success");
        // 登录成功，放入 session
        session.setAttribute("loginUser", loginUser);
    }
    return "login";
}
```

注意表单文本域中的 th:value 属性，如果 loginUser 不为 null，才会输出 userName 和 password 属性。

接下来，我们使用 Thymeleaf、Spring Cloud 等技术来实现一个简单的案例。

12.4 图书管理案例

本节的案例并不是要做一个多复杂的系统，而是希望通过这个案例，为大家提供一个研发的技术方案。

12.4.1 运行案例

如果需要运行本案例，请依次进行以下工作：

- 在 MySQL 中新建一个名称为 SALE_CLOUD 的数据库。
- 将 codes\12 目录下的 SALE_CLOUD.sql 导入 SALE_CLOUD 数据库。
- 配置数据库连接（如果 MySQL 的 root 用户密码为 123456 则无须修改）：打开 codes\12\run 目录下的 sc-user-service-0.0.1-SNAPSHOT.jar 和 sc-book-service-0.0.1-SNAPSHOT.jar，打开 BOOT-INF\classes 下的 application.yml 文件，修改 datasource 的几个数据库连接属性。
- 进入 codes\12\run 目录，按照以下顺序执行 bat 文件：eureka-server.bat、

book-service.bat、user-service.bat、gateway.bat、manage-web.bat、sale-web.bat。

- 在浏览器中访问后台登录界面 http://localhost:8110/manage-web/login，默认的用户名为 crazyit，密码为 123456。
- 前台展示界面的访问地址为 http://localhost:8210/sale-web/book/list。

12.4.2 案例模块

本案例所涉及的 Maven 模块较多，以下是对各模块的描述。

- sc-parent：项目的父模块，主要用于存放公共的依赖。
- sc-commons：保存映射的实体、工具类等。
- sc-book-service：Eureka 客户端，提供图书管理服务。
- sc-user-service：Eureka 客户端，提供用户服务，本案例只实现了用户验证服务。
- sc-gateway：集群网关。
- sc-eureka-server：Eureka 服务器。
- sc-manage-web：后台管理的表现层模块。
- sc-sale-web：前端销售的表现层模块。

以上模块的源代码均可在 codes\12\case 目录下找到。

12.4.3 案例架构

案例的架构如图 12-3 所示。

本节的案例主要涉及两个业务模块：图书管理和用户，它们会通过 Zuul 网关对外提供图书的管理、用户登录等服务。在表现层，会有销售前端和管理后台，在本案例中，管理后台只实现添加图书，销售前端负责展示图书。

注意：

案例中的“销售前端”和“管理后台”两个表现层模块，都不需要注册到 Eureka 中。

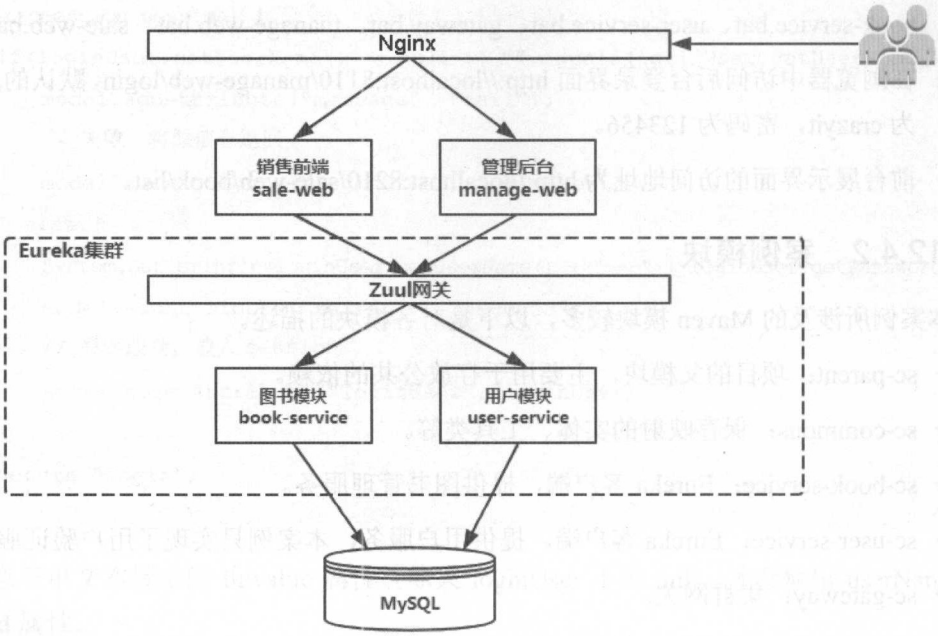


图 12-3 案例架构

12.4.4 数据库

本案例使用 MySQL 数据库，只需建立两个表：用户表和图书表，读者可以在 codes\12 目录下找到 SALE_CLOUD.sql 的脚本，导入数据库即可。用户表包括用户名、密码等字段；图书表主要包括名称、作者等常规字段。为了简单起见，笔者将图书的封面图片，以 blob 类型进行保存，在实际应用中，建议只保存图片的 URL。

12.4.5 用户登录

用户登录的页面保存在表现层模块 sc-manage-web 中，用户发送登录请求时，会向 sc-manage-web 的 Controller 提交表单，Controller 会调用 Feign 客户端，最终会调用 sc-user-service 所发布的验证服务，调用过程请见图 12-4。

服务集群的网关（sc-gateway 项目）需要配置路由转发，请见以下转发配置：

```
zuul:
  routes:
    book:
      path: /book/**
```

```
      serviceId: sc-book-service
    user:
      path: /user/**
      serviceId: sc-user-service
```

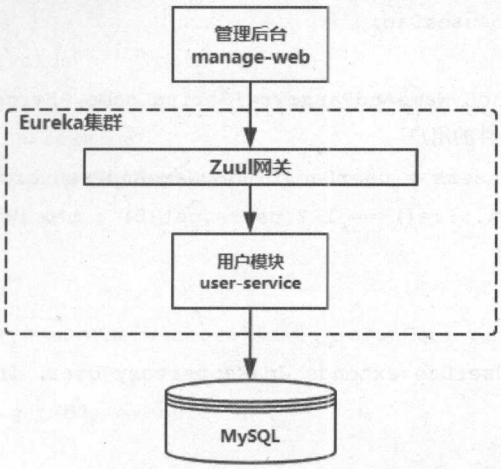


图 12-4 用户登录请求过程

在以上调用过程中，用户模块 sc-user-service 所发布的服务涉及 Controller、Service、Dao 等实现，请见代码清单 12-6。

代码清单 12-6：
codes\12\case\sc-parent\sc-user-service\src\main\java\org\crazyit\test\ctl\UserController.java
codes\12\case\sc-parent\sc-user-service\src\main\java\org\crazyit\test\service\UserService.java
codes\12\case\sc-parent\sc-user-service\src\main\java\org\crazyit\test\dao\UserDao.java

```
@RestController
public class UserController {

    @Autowired
    private UserService userService;

    @RequestMapping(value = "/validate/{name}/{password}", method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON_VALUE)
    public User validate(@PathVariable String name, @PathVariable String password) {
        return userService.findByNameAndPassowrd(name, password);
    }
}
```

```
@Service
public class UserService {

    @Autowired
    private UserDao userDao;

    public User findByNameAndPassowrd(String name, String password) {
        // 查询符合条件的用户
        List<User> users = userDao.findByNameAndPassword(name, password);
        return users.size() == 1 ? users.get(0) : new User();
    }
}

public interface UserDao extends JpaRepository<User, Integer> {

    /**
     * 根据名称和密码查询用户
     */
    List<User> findByNameAndPassword(String name, String password);
}
```

在表现层模块（sc-manage-web）中，使用 Feign 来调用用户模块发布的服务，sc-manage-web 模块的 Controller、Feign 客户端实现，请见代码清单 12-7。

代码清单 12-7：

codes\12\case\sc-parent\sc-manage-web\src\main\java\org\crazyit\test\service\UserService.java
codes\12\case\sc-parent\sc-manage-web\src\main\java\org\crazyit\test\ctl\ManageController.java

```
@FeignClient(name = "user", url = "http://localhost:9000/user")
public interface UserService {

    @RequestMapping(method = RequestMethod.GET, value = "/validate/{name}/
        {password}")
    User validate(@PathVariable("name") String name,
            @PathVariable("password") String password);
}

@Controller
```

```
@RequestMapping("/manage-web")
public class ManageController {

    @Autowired
    private UserService userService;

    @RequestMapping(value = "/login")
    public String login(@ModelAttribute User loginUser, Model model,
            HttpSession session) {
        if (loginUser.getName() == null) {
            return "login";
        }
        // 调用验证服务
        User dataUser = userService.validate(loginUser.getName(),
                loginUser.getPassword());
        if (dataUser.getId() == null) {
            // 验证失败
            model.addAttribute("message", "fail");
            return "login";
        } else {
            session.setAttribute("manage-user", loginUser);
            return "index";
        }
    }
}
```

UserService 是一个 Feign 客户端，本例中的 sc-manage-web 模块并不需要注册到 Eureka 服务器，因此 Feign 仅作为一个普通的 REST 客户端来使用，读者也可以使用其他的 REST 客户端。登录模板中的表单，请见代码清单 12-8。

代码清单 12-8：codes\12\case\sc-parent\sc-manage-web\src\main\resources\templates\login.html

```
<form th:action="@{/manage-web/login}" th:object="${loginUser}" method="post">
    <table>
        <tr>
            <td>用户名：</td>
            <td><input type="text" name="name" th:value="${loginUser?.name}"/> </td>
        </tr>
```

```
        <tr>
            <td>密码：</td>
            <td><input type="text" name="password" th:value="${loginUser?.password}"/>
            </td>
        </tr>
        <tr>
            <td> </td>
            <td><input type="submit" value="确定" class="input_button8"/></td>
        </tr>
    </table>
</form>
```

登录模块已经涵盖了全部技术，包括 Thymeleaf、Spring Data、Zuul、Feign 等，下面实现的功能将会简化对表现层实现的描述。

12.4.6 新建图书

新建图书的表单使用 Thymeleaf 的属性，表单将会被提交到 sc-manage-web（表现层）进行处理，该表现层模块会调用 sc-book-service 模块的服务。sc-book-service 会将图书记录保存到数据库中。服务模块 sc-user-service 提供的服务实现，请见代码清单 12-9。

代码清单 12-9：

codes\12\case\sc-parent\sc-book-service\src\main\java\org\crazyit\test\ctl\BookController.java

```
@RestController
public class BookController {

    @Autowired
    BookService bookService;

    @RequestMapping(value={"/save"}, method=RequestMethod.POST,
            consumes = MediaType.APPLICATION_JSON_VALUE)
    public void save(@RequestBody Book book) {
        bookService.save(book);
    }
}
```

在表现层（sc-manage-web）模块中，使用 Feign 调用网关服务，Controller、Feign 客户端的实现，请见代码清单 12-10。

代码清单 12-10：

codes\12\case\sc-parent\sc-manage-web\src\main\java\org\crazyit\test\service\BookService.java
codes\12\case\sc-parent\sc-manage-web\src\main\java\org\crazyit\test\ctl\ManageController.java

```
@FeignClient(name = "book", url = "http://localhost:9000/book")
public interface BookService {

    @RequestMapping(method = RequestMethod.POST, value = "/save")
    public void save(Book book);
}

@Controller
@RequestMapping("/manage-web")
public class ManageController {

    @Autowired
    private BookService bookService;

    @RequestMapping(value = "/book/save")
    public String saveBook(@ModelAttribute Book book, MultipartFile file)
            throws Exception {
        book.setCover(file.getBytes());
        // 保存数据
        bookService.save(book);
        return "forward:/manage-web/book/list";
    }
}
```

新建图书时，涉及图片上传。新建图书的表单，请见代码清单 12-11。

代码清单 12-11：

codes\12\case\sc-parent\sc-manage-web\src\main\resources\templates\book\add.html

```
<form method="post" th:action="@{/manage-web/book/save}"
    th:object="${book}" enctype="multipart/form-data">
    <table>
        <tr>
          <td>书名：</td>
          <td><input name="name" type="text"/></td>
        </tr>
```

```
        <tr>
          <td>作者：</td>
          <td><input name="author" type="text"/></td>
        </tr>
        <tr>
          <td>价格：</td>
          <td><input name="price" type="text"/></td>
        </tr>
        <tr>
          <td>图片：</td>
          <td>
            <input type="file" name="file">
          </td>
        </tr>
    </table>
    <input type="submit" value="确定"/>
</form>
```

在实际应用时，涉及图片的话，最好使用文件服务器进行保存，数据库中只保存图片的URL，这样可减少数据传输所占用的带宽。

12.4.7 图书展示

图书展示功能较为简单，只需要调用sc-book-service的服务，查找全部Book数据即可，在此不过多讲述，仅将表现层的Controller的实现描述一下，请见代码清单12-12。

代码清单12-12：

codes\12\case\sc-parent\sc-sale-web\src\main\java\org\crazyit\test\controller\SaleWebController.java

```
@RequestMapping(value = "/book/list")
public String listBook(Model model, HttpServletRequest request) {
    List<Book> books = bookService.getBooks();
    // 写到临时文件中
    for (Book book : books) {
        String coverUrl = ImageUtil.writeToImage(book, request);
        book.setCoverUrl(coverUrl);
    }
    model.addAttribute("books", books);
```

```
        return "book/list";
    }
```

唯一需要注意的是，本例的图书封面图片会保存到数据库中。因此在读取时，ImageUtil.writeToImage 方法会将 byte 数组存放到临时文件中，再获取文件的地址，设置到 Book 对象中。模板页面实现较为简单，直接遍历集合，输出属性值即可：

```
<li th:each="data : ${books}">
    <img th:src="${data.coverUrl}" src="images/book.png" />
    <div>
        <span th:text="${data.name}"> </span>
    </div>
    <div>
        价格：<span th:text="${data.price}"></span>
    </div>
    <div>
        作者：<span th:text="${data.author}"> </span>
    </div>
</li>
```

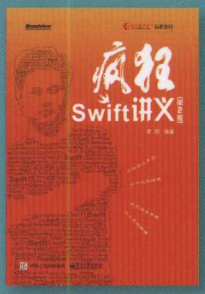
12.5 本章小结

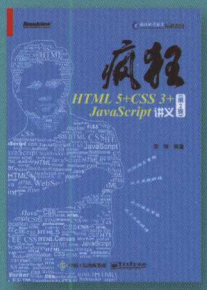
在本章的前半部分介绍了 JSP 与 Thymeleaf 两个表现层技术。学习完本章的 12.2 节后，读者可以整合 Spring Boot 与 JSP。在 12.3 节，粗略地讲述了模板引擎 Thymeleaf，对于一些常用的功能列举了使用的例子。读者根据这些例子，可以很快熟练使用 Thymeleaf。如果要更深入学习 Thymeleaf，还需要读者自行查阅 Thymeleaf 的文档。

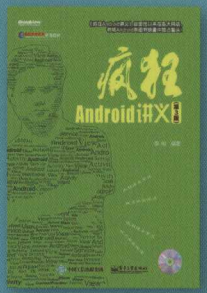
在本章的最后，以一个小案例来使用本书所涉及的主要技术，例如 Spring Cloud 的相关框架、Spring Data、Thymeleaf 等。虽然案例功能简单，但读者可以以该案例的技术为基础进行扩展，以应用到实际的项目中。

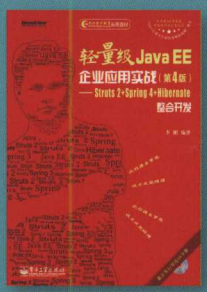
Li Gang's Series

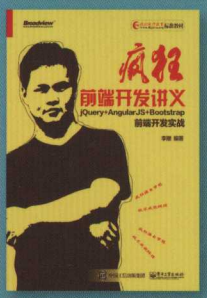
李刚 作品系列











Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

疯狂 Spring Cloud 微服务架构实战

作者杨恩雄，开源中国博主，发表的Spring Cloud以及Activiti的相关技术文章极具参考价值，帮助不少用户解决了实际问题。这本《疯狂Spring Cloud微服务架构实战》的内容由浅到深，原理与实战相结合，可以帮助初学者，甚至是具有一定微服务知识的开发者，快速掌握Spring Cloud的相关知识。

红薯（www.oschina.net）开源中国创始人

Spring Cloud提供了一套完整的微服务解决方案，为企业IT架构变革和发展保驾护航。《疯狂Spring Cloud微服务架构实战》一书由浅入深、抛砖引玉，讲解了各个组件的使用方法，可帮助开发者快速开发并上线微服务应用。

许进（xujin.org）Spring Cloud中国社区创始人

与杨恩雄共事多年，他拥有较强的总结能力和技术实力，出版了不少与Java相关的书籍。此次出版的《疯狂Spring Cloud微服务架构实战》一书，全面介绍了Spring Cloud的主要框架，为Java应用开发提供了一整套解决方案。开发者学习完这本书中的知识后，我相信他们在技术实力上会有一个质的提升。

钟永生 YY欢聚时代技术经理

阅读此书有任何技术问题或欲获取本书配套资源，可登录http://www.crazyit.org
亦可访问http://www.broadview.com.cn/33109下载本书配套资源。

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

策划编辑：张月萍
责任编辑：刘　舫
封面设计：侯士卿

上架建议：计算机/微服务

ISBN 978-7-121-33109-1

定价：58.00元